REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	6 June 1997	Final Progress Report 15 March 94 to 14 March 97

4. TITLE AND SUBTITLE
Efficient Parallel Semantic/O-O Database Management

5. FUNDING NUMBERS
DAAH04-94-G-0024

6. AUTHORS
Naphtali Rishe

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Florida International University
School of Computer Science
University Park, ECS 354
Miami, FL 33199

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
ARO 32427.32-MA-SDI

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
We have developed an optimistic concurrency control method for a massively parallel semantic database machine. Our concurrency control algorithm achieves very fine granularity, ensures serializability and external consistency, and uses local logical clocks which do not require physical clock synchronization. We have also developed a dynamic load balancing algorithm which repartitions data among processors using a fault-tolerant data transfer policy to produce a more evenly balanced load. We have implemented benchmarks on our experimental semantic database system that have shown it to be more than competitive with current commercial products. In addition to these results, we have continued to perform research on semantic databases. Our research into applying SQL to semantic databases has shown the advantages of the semantic binary model even when using standard relational languages.

14. SUBJECT TERMS
semantic database, database machine, concurrency control, load balancing, benchmarks

15. NUMBER OF PAGES
43

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-1
298-102

TITLE: EFFICIENT PARALLEL SEMANTIC/O-O DATABASE MANAGEMENT

FINAL PROGRESS

NAPHTALI RISHE

DATE: 6 June 1997

U.S. ARMY RESEARCH OFFICE

CONTRACT/GRANT NUMBER: DAAH04-94-G-0024

FLORIDA INTERNATIONAL UNIVERSITY

TABLE OF CONTENTS

1. LIST OF APPENDICES

- *Efficient Optimistic Concurrency Control in Massively Parallel B-trees with Variable Length Keys*
- *Load Balancing in a Massively Parallel Semantic Database*
- *SB2 Benchmark (Consumer Survey Database)*
- *Semantic SQL*

2. FINAL PROGRESS REPORT

2.1. Statement of the Problem Studied

Demands for databases with high efficiency and high throughput have led to the recent development of several types of database systems utilizing parallel processors. This project involved research aiming to improve the state of the art of highly parallel database systems. Both the logical properties (usability) and the physical properties (efficiency) were enhanced. We developed very efficient algorithms for parallel database management systems in semantic/object-oriented models. Our approach has several advantages over the currently known theory and results on database machines:

- Unlike the current database machines based on the contemporary Relational Model of databases, our work is based on semantic data models (including, in the broad sense, object-oriented models as well as storage of multi-media data). The use of semantic models assures better logical properties: friendlier and more intelligent user interfaces, comprehensive enforcement of integrity constraints, improved database design, greater flexibility, and substantially shorter application programs (which reduces the programming effort and facilitates program verification).
- At the physical level, the system is more efficient than existing database systems. The algorithms and prototype system developed are highly-efficient. In particular, the use of the semantic model allows better exploitation of the parallelism.

This project focused on two problems: concurrency control and load balancing.

2.2. Summary of the Most Important Results

2.2.1. Concurrency control

We have developed an improved semantic optimistic concurrency control algorithm and a query optimization technique (lazy queries) that can be used in a massively parallel B-tree with variable-length keys. B-trees with variable-length keys can be effectively used in a variety of database types. In particular, we are using this B-tree structure, which also offers data compression, in our implementation of a semantic object-oriented DBMS. Our concurrency control algorithm uses semantically safe optimistic virtual "locks" that achieve very fine granularity in conflict detection. Our algorithm contributes smaller transaction conflict probability by using very fine granularity (attribute or string level granularity). We have proven that the algorithm ensures serializability and external consistency. Our algorithm uses local logical clocks and does not require physical clock synchronization. A

lazy query execution algorithm is used to reduce the client-server traffic and improve the granularity of concurrency control by minimizing the number of optimistic locks. Most relevant processing is done at the client machines, thus reducing the data and processing overheads at parallel B-tree servers.

Our concurrency control scheme is detailed in the appendix entitled *Efficient Optimistic Concurrency Control in Massively Parallel B-trees with Variable Length Keys.*

2.2.2. Load balancing

Load balancing is essential in our system to equalize the data and transactional load among B-tree partitions. In our system, load balancing is performed as a series of load balancing transactions that transfer small string intervals from one partition to another. Load balancing transactions require a large amount of resources and time to move the data. Thus, the load balancing transactions should be well coordinated in the system. Since the data movement time is large, a centralized algorithm performs well without becoming a system bottleneck. In our B-tree, a centralized load balancing module periodically collects data and load distribution statistics from all partitions and heuristically generates a data distribution policy, initiating the load balancing transactions executed at the B-tree partitions. Apart from the load balancing transactions, the load balancing module can create a new partition or delete an empty partition which is no longer referred to by any other B-tree partition. Our concurrency control algorithm maintains its safety and efficiency notwithstanding load balancing activity performed according to our algorithm.

Our load balancing scheme is detailed in the appendix entitled *Load Balancing in a Massively Parallel Semantic Database.*

2.2.3. Benchmarks

We have developed benchmarks for the semantic database engine. The purpose of the benchmarks is to show that while our SDB offers substantially better flexibility and logical properties, its performance is not much worse, and is often much better than that of the best other DBMS's.

Our first semantic benchmark, SB1, run on a Pentium-200 computer with 128MB of RAM, showed that on certain types of queries the Semantic Database is 30 times faster than a highly-optimized fully-indexed Oracle database working on the same hardware. The Oracle database also requires about 10 times more disk space than SDB. Using another variation of the schema of the Oracle database, we managed to reduce Oracle space requirements to about 3 times more than that of SDB, but then the speed of Oracle was up to 120 times slower than SDB. Although the results are already quite favorable to SDB, we expect to further improve (reduce) the space requirements of SDB by implementing our new data compression algorithms in the next SDB release.

Our second semantic benchmark, SB2, run under similar circumstances, has shown similar results. The preliminary results of the SB2 benchmark are presented in the appendix entitled *SB2 Benchmark (Consumer Survey Database).*

The tests have also shown that Oracle requires a lot of fine-tuning to make it work fast on a particular benchmark. The Semantic database does not require any tuning at all, which means that the same installation of SDB will work equally well on all databases that are used

on a server. Oracle, being tuned for one application, will not perform as well for another application, nor will it perform as well for the same application when users pose new types of ad-hoc queries.

2.2.4. Semantic SQL

While not directly supported under this project, a related project supported by funding leveraged from this project has resulted in the adaptation of SQL (Structured Query Language), which is the standard language for relational databases, to semantic databases. The original purpose of this adaptation was to be compatible with and be able to communicate with relational tools. However, it turned out that the size of a typical SQL program for a semantic database is many times smaller than for an equivalent relational database. SQL offers significant benefits when used with semantic databases and although originally intended for relational databases, offers many advantages to the semantic model. We have implemented a stand-alone SQL server as well as an embedded-SQL preprocessor.

A multi-user semantic database engine has been developed and is now in the testing phase. A user interface to this engine has been developed using C++ and is also in the testing phase. Our ODBC driver for the semantic database engine is now fully operational and allows SQL querying of a semantic database and interoperability with relational database tools such as Microsoft Access's Query-by-Example tool. Using these tools, the number of user keystrokes required is in correlation to the size of the generated SQL program. Since the SQL programs for the semantic database are substantially shorter, the third-party query tools are much more ergonomic with the semantic database than with the relational databases for which they were originally designed.

Semantic SQL is detailed in the appendix entitled *Semantic SQL.*

2.3. List of All Publications and Technical Reports

- G. Cao, N. Rishe. "A Nonblocking Consistent Checkpointing Algorithm for Distributed Systems." Proceedings of the IASTED Eighth International Conference Parallel and Distributed Computing and Systems October 16-19, 1996 - Chicago, Ill., pp 302-307.
- C. Chen, N. Rishe. "Channel Allocation for Queries Over Integrated Services Digital Networks." Florida International University School Of Computer Science Technical Report #96-03.
- C. Liu, A. Ouksel, P. Sistla, J. Wu, C. Yu, and N. Rishe. "G-tree: Performance evaluation and application in Fuzzy databases", International Conference on Information and Knowledge Management 1996.
- K. L. Liu, G. J. Lipovski, C. Yu, N. Rishe. "Efficient Processing of One and Two Dimensional Proximity Queries in Associative Memory." Proceedings of ACM SIGIR 1996, pp 138-146.
- W. Meng, C. Yu, W. Wang and N. Rishe. "Performance Analysis of Three Text-join Algorithms." IEEE Transactions on Knowledge and Data Engineering, in press.

- W. Meng, C. Yu, W. Wang and N. Rishe. ''Performance Analysis of several algorithms for processing Joins between textual attributes.'' Proceedings of the IEEE International Data Engineering Conference, 1996, pp. 636-644.
- C. Orji, N. Rishe, K. Nwosu. ''Data Layout for Interactive Video-on-Demand Storage Systems.'' Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering (SEKE '96), Nevada, June 1996, pp 285-292.
- C. Orji, N. Rishe, K. Nwosu. ''Dynamic Reallocation of Multimedia Data Objects.'' Proceedings of the Pacific Conference on Distributed Multimedia Systems, Hong Kong, pp 163-170.
- C. Orji, N. Rishe, K. Nwosu. ''Multimedia Object Storage and Retrieval.'' Proceedings of the International Symposium on Multimedia Systems, Yokohama, Japan, March 1996, pp 368-375.
- N. Rishe. ''Managing Network Resources for Efficient, Reliable Information Systems,'' panel position paper, *Proceedings of the Third International Conference on Parallel and Distributed Information Systems* (Austin, Texas, September 28-30, 1994), IEEE Computer Society Press, 1994, pp. 223-226.
- N. Rishe. ''A Universal Model for Non-procedural Database Languages.'' Fundamenta Informaticae, April 1996, pp 31-57.
- N. Rishe, D. Barton, M. Sanchez. ''Storage and Visualization of Spatial Data in a High-performance Semantic Database System.'' Florida International University Technical Report #95-15.
- M. Sanchez, N. Rishe. ''Dynamic Bandwidth Allocation for an ISDN WAN.'' Florida International University School Of Computer Science Technical Report #96-02.
- N. Rishe, A. Shaposhnikov, and W. Sun. ''Load Balancing Policy in Massively Parallel Semantic Database'' Proceedings of the First International Conference on Massively Parallel Computing Systems (Ischia, Italy, May 2-6, 1994), IEEE Computer Society Press, 1994, pp. 328-333.
- N. Rishe, A. Shaposhnikov, S. Graham. ''Load Balancing in a Massively Parallel Semantic Database.'' *Computer Systems Science and Engineering* (Special issue on massively parallel processing), 11, *4* (July 1996), pp. 195-199.
- N. Rishe and W. Sun. ''A pipeline CASE tool for database design.'' Proceedings of SEKE'94: Sixth International Conference on Software Engineering and Knowledge Engineering, pp. 336-343. KSI, 1994.
- N. Rishe, W. Sun, D. Barton, Y. Deng, C. Orji, M. Alexopoulos, L. Loureiro, C. Ordonez, M. Sanchez, A. Shaposhnikov. ''Florida International University High Performance Database Research Center.'' *SIGMOD Record*, 24 (1995), 3, pp. 71-76.
- N. Rishe and O. Zhukov. ''High Performance Computer Technology for Solving Complex Physics Problems'', International School-Seminar on Automation and Computing in Nuclear Physics and Astrophysics, Yalta, 1996.
- M. Sanchez, D. Barton, N. Rishe. ''Application of a High Performance Semantic Database to GIS Data Requirements.'' Proceedings of the Eleventh Annual Louisiana

Remote Sensing and Geographic Information Systems. April 18-20, 1995.

- M. Sanchez, D. Barton, N. Rishe. ''Semantic Database for Geographic Information Systems.'' Proceedings of the IEEE Southeast Conference. April 11-14, 1996, pp 696-698.
- M. Sanchez, C. Orji, K. Nwosu, N. Rishe. ''Time Mechanics as Applied to Event Ordering.'' Proceedings of the IEEE Southeast Conference. April 11-14, 1996, pp 661-664.
- M. Sanchez, N. Rishe, D. Barton. ''Specialized GIS Via a High Performance Semantic Database.'' Proceedings of the 12th Annual Louisiana Remote Sensing and Geographic Information Systems Workshop. April 16-18, 1996. Invited Paper.
- L. Yan, W. Sun, N. Prabhakaran, S. Guo, Y. Deng, N. Rishe. ''A Dynamic Hypermedia Model for Interactive Video.'' Proceedings of 13th International Conference on Advanced Science and Technology in conjunction with the 2nd International Conference on Multimedia Information Systems. April 97, Illinois.
- O. Zhukov, N. Rishe, C. Ordonez. ''An Approach to Building a Highly Parallel Computer System.'' Florida International University School of Computer Science Technical Report #95-3.

2.4. List of All Participating Scientific Personnel Showing Any Advanced Degrees Earned By Them While Employed on the Project

Faculty: Naphtali Rishe

Other Personnel: U. Alfaro, M. Chekmasov, M. Chekmasova, M. Drobintsev, E. Ekanayake, M. Ferreiro, S. Graham, E. Greywoode, R. Hanif, I. Kaprizkina, A. Kirienko, C. Luo, V. Mallampati, M. Monga, K. Naboulsi, V. Patil, D. Perednya, P. Rusconi, A. Shaposhnikov, D. Sroka, X. Su, J. Tang, S. Tucker, D. Vasilevsky, S. Wang, M. Woon Choy, J. Xu

Degrees Awarded: M.S.: E. Ekanayake, K. Naboulsi, V. Patil, S. Wang; B.S.: P. Rusconi

3. REPORT OF INVENTIONS (BY TITLE ONLY)

- ''Efficient Optimistic Concurrency Control in Massively Parallel B-trees.'' Status: patent application has been prepared, about to be submitted.
- ''Parallel Semantic DBMS.'' Status: invention disclosed to the University; entered into agreement on sharing rights.

4. APPENDICES

Efficient Optimistic Concurrency Control in Massively Parallel B-trees with Variable Length Keys[1]

Naphtali Rishe and Artyom Shaposhnikov

High-performance Database Research Center
School of Computer Science
Florida International University
University Park, Miami, FL 33199
Telephone: (305) 348-2025, 348-2744
FAX: (305)-348-3549; E-mail: {rishen, shaposhn}@fiu.edu

Abstract

This paper proposes an efficient optimistic concurrency control algorithm and a query optimization technique (lazy queries) used in a massively parallel B-tree with variable-length keys. B-trees with variable-length keys can be effectively used in a variety of database types. In particular, we show how such a B-tree is used in our implementation of a semantic object-oriented DBMS. Our concurrency control algorithm uses semantically safe optimistic virtual "locks" that achieve very fine granularity in conflict detection. We prove that the algorithm ensures serializability and external consistency. Our algorithm uses local logical clocks and does not require physical clock synchronization. Lazy query execution algorithm is used to reduce the client-server traffic and improve the granularity of concurrency control by minimizing the number of optimistic locks. Most relevant processing is done at the client machines, thus reducing the data and processing overheads at parallel B-tree servers.

1. Introduction

B-tree data structures are widely used in implementation of databases. B-trees allow to insert, delete, find, and retrieve a number of database records [Comer-79]. With the advent of parallel databases that can store thousands of terabytes of data and object-oriented technology, several new properties of B-tree data structures are highly desirable:

- Transparent access to massive volumes of data. To index data database systems use keys or object identifiers. A B-tree record is usually divided into two parts: an index part and a data part. The B-tree records are indexed lexicographically by the index part. The index part in B-trees is usually of fixed size and the database capacity is limited. A new generation of databases (for example, [Rishe-92-DDS]) do not have keys at all: the data itself serves as a key of varying length. In a semantic database implementation, for example, each entity comprises a large number of strings, each of which corresponding to an attribute or relationship.
- On-line transaction processing systems demand high transactional and query throughput. Such throughput requires running many transactions and queries in parallel. Efficient

[1]This research was supported in part by NASA (under grant NAGW-4080) and ARO (under BMDO grant DAAH04-0024).

concurrency control and query optimization algorithms are necessary to resolve conflicts between concurrent transactions.

In this paper we propose algorithms of a new B-tree structure that combines the following properties:

- Parallel multicomputer operation. We employ fast optimistic concurrency control. Granularity at the level of strings is attained. However, there is no overhead in the physical data structure.
- Semantically safe optimistic locks. We achieve greater degree of safety in transaction conflict detection than in many algorithms that use locking. Even if transaction relies on absence of some data in the B-tree, and this data was inserted by another concurrent transaction, a conflict will be detected.
- Efficient query execution algorithm – lazy queries – that gives faster execution of complex queries and better granularity in concurrency control.
- Transparent variable size keys. In our B-tree, the whole record is a key (it is up to the B-tree client how to divide this key into index and data portion, if that is necessary). A record in our B-tree is called a *string*.
- String data compression. All strings in data and index blocks in our B-tree are compressed by eliminating common prefixes. Additionally, the index strings are compressed by eliminating redundant suffixes. This results in very short index strings. Index compression not only reduces the storage requirements but also accelerates the B-tree operations by keeping more index data in a memory cache.
- Automatic background load balancing that redistributes the data among the database computers to equalize the data and transactional load.
- Very high capacity. The size of our parallel B-tree is virtually unlimited (specifically, it is 10^{36} bytes for 64 bit computers).

The rest of this paper is organized as follows. Section 2 defines the logical level of our B-tree. In Section 3 we describe a semantic database application that utilizes our B-tree and show some typical database queries. Section 4 describes a query optimization technique "lazy queries" that we use to reduce the number of server accesses and transaction conflict probability. A parallel B-tree architecture is presented in Section 5. Our concurrency control algorithm is presented in Section 6.

2. Elementary B-tree operations

We define B-tree as an implementation of a data type, each instance of which is a lexicographic ordered set of strings with the following operations:

1. Elementary query (interval) operator $[l, r]$, where l and r are arbitrary strings.

   $[l, r]\ \mathbf{S} = \{ x \in \mathbf{S} \mid l \le x \le r \}$, where $\le$ is the lexicographic order of strings.

2. Update operator. Let **D** and **I** be disjoint sets of strings:

   $\mathbf{S} + (\mathbf{I},\mathbf{D}) = (\mathbf{S} - \mathbf{D}) \cup \mathbf{I}$ (I.e. we remove a set of strings **D** and insert a set **I** instead).

The next section describes how these elementary operations were used in the implementation of semantic binary object-oriented database.

3. Sample Application: Semantic DBMS Implementation

The semantic database models in general, and the Semantic Binary Model SBM ([Rishe-92-DDS] and others) in particular, represent the information as a collection of elementary facts categorizing objects or establishing relationships of various kinds between pairs of objects. The central notion of semantic models is the concept of an abstract object, which is any real world entity that we wish to store information about in the database. The objects are categorized into classes according to their common properties. These classes, called categories, need not be disjoint, that is, one object may belong to several of them. Further, an arbitrary structure of subcategories and supercategories can be defined. The representation of the objects in the computer is invisible to the user, who perceives the objects as real-world entities, whether tangible, such as persons or cars, or intangible, such as observations, meetings, or desires.

The database is perceived by its user as a set of facts about objects. These facts are of three types: facts stating that an object belongs to a category; facts stating that there is a relationship between objects; and facts relating objects to data, such as numbers, texts, dates, images, tabulated or analytical functions, etc. The relationships can be of arbitrary kinds; for example, stating that there is a many-to-many relation address between the category of persons and texts means that one person may have an address, several addresses, or no address at all.

Logically, a semantic database is a set of facts of three types: categorization of an object: **xC**, relationship between two objects: **xRy**, relationship between an arbitrary object and a value: **xRv**. Efficient storage structure for semantic models has been proposed in [Rishe-91-FS]. The collection of facts forming the database is represented by a file structure which ensures approximately one disk access to retrieve any of the following:

1. For a given abstract object **x**, verify/find what categories the object belongs to.
2. For a given category, find its objects.
3. For a given abstract object **x** and relation **R**, retrieve all **y** such that **xRy**.
4. For a given abstract object **y** and relation **R**, retrieve all abstract objects **x** such that **xRy**.
5. For a given abstract object **x**, retrieve (in one access) all (or several) of its categories and direct and/or inverse relationships, i.e. relations **R** and objects **y** such that **xRy** or **yRx**. The relation **R** in **xRy** may be an attribute, i.e. a relation between abstract objects and values.
6. For a given relation (attribute) **R** and a given value **v**, find all abstract objects such that **xRv**.
7. For a given relation (attribute) **R** and a given range of values $[\mathbf{v}_1, \mathbf{v}_2]$, find all objects **x** and **v** such that **xRv** and $\mathbf{v}_1 \le \mathbf{v} \le \mathbf{v}_2$.

We call the operations 1 through 7 *elementary queries*. The entire database can be stored in a single B-tree. This B-tree contains all of the facts of the database (**xIC, xRv, xRy**) and also additional information called inverted facts: **CIx**, **Rvx,** and $\mathbf{yR}_{inv}\mathbf{x}$ (Here, **I** is the pseudo-relation IS-IN denoting membership in a category). The inverted facts allow to keep answers to the queries 2, 4, 6, 7 in a contiguous segment of data in the B-tree and answer them with one disk access (when the query result is much smaller than one disk block). The direct facts **xIC**

and **xRy** allow to answer the queries 1, 3, and 5 with one disk access. This allows both sequential access according to the lexicographic order of the items comprising the facts and the inverted facts, as well as random access by arbitrary prefixes of such facts and inverted facts. The facts which are close to each other in the lexicographic order reside close in the B-tree. (Notice, that although technically the B-tree-key is the entire fact, it is of varying length and on the average is only several bytes long, which is the average size of the encoded fact **xRy**.).

Numeric values in the facts are encoded as substrings using the order-preserving variable-length number encoding of [Rishe-91-IB].

Query	B-tree Implementation
1. **x?**	**[xI, xI+ 1]**
2. **C?**	**[CI, CI+ 1]**
3. **xR?**	**[xR, xR + 1]**
4. **?Rx**	**[xR_{inv}, xR_{inv}+ 1]**
5. **x??**	**[x, x + 1]**
6. **?Rv**	**[Rv, Rv + 1]**
7. **R[v_1..v_2]?**	**[Rv_1, Rv_2 + 1]**

Table 1. Implementation of elementary queries

Table 1 summarizes how the elementary semantic queries are implemented using the B-tree interval operators. We use notation **S + 1** to denote a string derived from the original string **S** by adding 1 to the last byte of **S** (For strings encoding abstract objects, this operation never results in overflow).

For most elementary queries (Queries 1, 3, 4, 5, 6) the number of binary facts is usually small. Some queries (Queries 2 and 7), however, may result in a very large number of facts and it may be inefficient to retrieve the whole query at once.

A common operation in databases is to calculate an intersection of two queries. For example, consider a query: "Find all objects from category Student that have the attribute BirthYear 1980". This query can be executed using several scenarios:

Scenario 1.

a. Retrieve all persons born in 1980: execute an elementary query "BirthYear 1980 ?"

b. For each person retrieved in the step *a* verify that the person belongs to the category Student

Scenario 2.

a. Retrieve all persons born in 1980: execute an elementary query "BirthYear 1980 ?"

b. Retrieve all students: execute an elementary query "Student ?"

c. Find an intersection of the objects retrieved in *a* and *b*.

In Scenario 1 we retrieve all persons from all categories (Person, Instructor, and Student) who were born in 1980 and for each person we execute an additional elementary query to verify that the retrieved person is a student. In this scenario we have to execute a large number of small queries.

In Scenario 2 we execute only two elementary queries and then find an intersection of the results. The problem is that the elementary query "Student ?" may result in a very large set of binary facts. Not only is this very inefficient in terms of expensive communication between client and server, but also such big query would be affected by any transaction that inserts or deletes students and our query would be aborted more often than the query in the Scenario 1.

Thus, Scenario 1 is obviously better in our case. Consider now another query: "Find all instructors born in 1970". The number of persons born in 1970 could be larger or comparable with the total number of instructors. In this case, Scenario 2 would be much more efficient because we need to execute only two elementary queries.

The next section introduces a technique of lazy elementary query execution that greatly reduces the number of disk accesses, the server traffic, and the transaction conflict probability by automatically reducing one scenario to another. For example, the intersection operator get a close-to-optimal implementation without keeping any data distribution statistics.

4. Lazy Queries

In our B-tree the actual query execution is deferred until the user actually requests the query results. We define the elementary lazy query programmatic interface in a B-tree **B** as follows:

1. **Q := [*l*, *r*] B** – define a lazy query **[*l*, r]** but do not execute it yet. Let **Q.P** be a pointer to future results of the query. Initially **Q.P^ := ''** , i.e. **P** points to an empty string.
2. ***Seek*(Q, *x*)** – Moves the pointer **Q.P**, so that **Q.P^ = min{*y* ∈ [*l*, *r*]B | *y* ≥ *x*}.**

The actual principal operations on the query results are derived from the above:

1. ***Read*(Q) := Q.P^** – reads the current string pointed by the logical pointer **Q.P**. This operation results in an error if **Q.P = null**.
2. ***Next(Q)* := *Seek*(Q, Read(Q) + 0).** We use notation ***s* + 0** to denote a string derived from the string ***s*** by appending a zero byte.

When the ***Seek*** operation is executed, the string pointed to by the new logical pointer is fetched from the B-tree and, normally, a small number of lexicographically close strings is prefetched and placed in a lazy query cache buffer. It is likely that the next ***Seek*** operation will request a string which is already in the cache buffer, so only a few ***Seek*** operations require actual disk and server access.

Many queries can efficiently use the ***Seek*** operation. For example, we can find the intersection of two lazy queries Q_1 and Q_2 very efficiently: construct a new lazy query (lazy intersection) Q_3 where the ***Seek*** operation uses algorithm shown in Figure 1.

```
Seek(Q3, y):

Seek(Q1, x);
Seek(Q2, x);
while (Q1.P ≠ null & Q2.P ≠ null &
       Q1.P^ ≠ Q2.P^) do
   if Q1.P^ > Q2^.P then
      Seek(Q2, Q1.P^)
   else
      Seek(Q1, Q2.P^);
od;
if Q1.P = null or Q2.P = null then
   Q3.P := null
else
   Q3.P := Q1.P;
```

Figure 1. Algorithm to find intersection $Q_3 := Q_1$ & Q_2

This algorithm gives an efficient solution for the sample queries described in the previous section. For the query "Find all objects from category Student that have the attribute BirthYear 1980" we use three lazy queries:

a. Q_1 := elementary lazy query "BirthYear 1980 ?"
b. Q_2 := elementary lazy query "Student ?"
c. $Q_3 := Q_1$ & Q_2

Since query Q_3 is not actually executed, our algorithm that finds intersection will not require to fetch every student from the database: the number of actual disk accesses to retrieve the students in the query Q_2 will be less than or equal to the number of persons born in 1980. Thus, the cost of the lazy query Q_3 will be smaller than the cost of the optimal solution for elementary queries in Scenario 1 described in the previous section.

For the query "Find all instructors born in 1970" we use three similar lazy queries. Since the number of instructors is likely to be small, it is possible that all instructors will be fetched in the first disk access, and the whole query will require a number of server accesses close to 2, which is the optimal number.

Lazy queries not only result in a smaller number of server accesses. We will show that lazy queries allow to improve the granularity of our concurrency control algorithm and reduce the transaction conflict probability.

5. Parallel B-tree

A massively parallel B-tree should perform many queries and transactions simultaneously and its size should scale to hundreds of terabytes even if the underlying computer hardware supports only 32 bit addressing. This can be achieved by splitting the B-tree into partitions of about 1 gigabyte in size. The whole B-tree is then a network of computers where each computer holds one or more B-tree partitions.

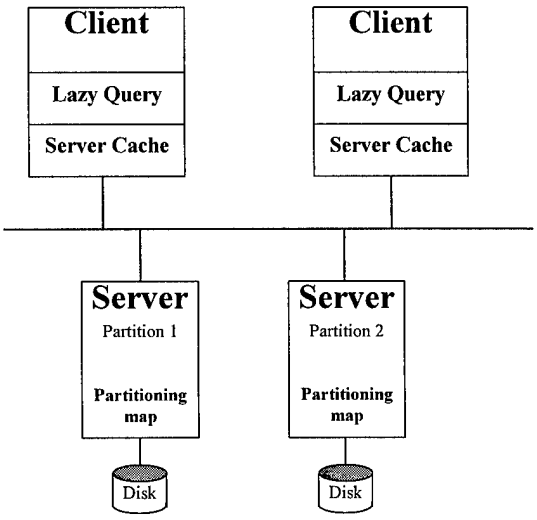


Figure 2: Client-server model of Parallel B-tree

The B-tree partitions themselves must be indexed. This index is also represented as a B-tree which is called a Partitioning Map. The Partitioning Map B-tree can reference approximately 200,000,000 B-tree partitions. Since each partition can hold up to about 1GB of data, the total addressable B-tree space is limited by 200,000 Terabytes. For 64 bit computers this limit becomes 10^{18} times larger, which is about 10^{36} bytes and is beyond any practical database size. Figure 1 shows typical client-server model of parallel B-tree.

6. Concurrency Control

Our concurrency control algorithm is an optimistic algorithm that first accumulates a transaction, and then performs it using a 2-phase commit protocol [Gray-79], and backward validation [Haerder-84] to ensure the serializability and external consistency of transactions. Our algorithm benefits from and improves upon the validation technique of the [Adya&al-95] algorithm for an object-oriented database. Their algorithm uses a loosely synchronized physical clocks to achieve global serialization and detects conflicts at the object level granularity. In our algorithm, a finer granularity at the level of strings is attained and we used logical clocks to achieve global serialization; nevertheless, our algorithm does not require maintaining any extra data per string or per client.

6.1. Accumulation of Transactions

In a parallel B-tree updates and queries made by a client should be verified for conflicts with updates and queries made simultaneously by the other B-tree clients. A transaction is a group of B-tree updates and queries which is guaranteed to be consistent with the queries and updates executed concurrently within other transactions. To create such a group of operations we have several B-tree operations in addition to the lazy queries defined in Section 4:

1. Insert String x
2. Delete String x
3. Transaction Begin
4. Transaction End

A transaction is a sequence of all lazy queries and updates (Operations 1,2) executed between the Transaction Begin and Transaction End. When the Transaction End is executed, all queries and updates made since the Transaction Begin are checked for conflicts with the queries and updates made by concurrent transactions. If there is a conflict, the transaction is aborted and the Transaction End returns an error.

The updates made within a transaction do not change the B-tree immediately. Instead, these updates are accumulated at the client side in a set of inserted strings **I** and a set of deleted strings **D**. The B-tree strings remain unaffected. The insert and delete operations work as follows:

$$\textbf{insert}(x) = \{\, \mathbf{D} := \mathbf{D} - \{x\};\ \mathbf{I} := \mathbf{I} \cup \{x\} \,\}$$
$$\textbf{delete}(x) = \{\, \mathbf{I} := \mathbf{I} - \{x\};\ \ \mathbf{D} := \mathbf{D} \cup \{x\} \,\}$$

When Transaction End is executed, the set **D** is deleted from the B-tree and the set **I** is inserted into B-tree:

$$\mathbf{B} := (\mathbf{B} - \mathbf{D}) \cup \mathbf{I}$$

During the accumulation of a transaction into sets **D** and **I**, our concurrency control algorithm at the client also accumulates a set **V** to be used for backward validation. The set **V** contains the specification of each subinterval read by a query within the transaction and a timestamp of this reading. A subinterval is a subrange within a query which subrange was physically retrieved from one database partition at one logical moment in time. The logical time at a given database partition is incremented every time a committed transaction physically changes that partition. The subintervals are stamped with this logical time and a number that identifies the partition in the system. Thus the set **V** is $\{([l_k, r_k], t_k, p_k)_{k=1}^{n}\}$, where t_k is the timestamp and p_k is the partition number.

In our validation technique, when committing a transaction T, the system does not need to remember the results of T's queries; it remembers only query specifications $[l, r]$, which are checked against concurrent transactions at T's commit time. The validation is done against transaction queues, normally without any disk access.

Lazy queries can be used to further reduce the validation specified by the set **V** and improve the granularity in conflict detection. Previous examples have shown that the user does not actually retrieve all facts from the lazy query interval. The intersection of lazy queries uses the ***Seek*** operation and actually retrieves only a few strings from the original elementary queries. In our implementation, a lazy query automatically keeps track of those string subranges that have actually been by the user. This union of subranges can be much smaller than the union of the original elementary query intervals. This results in a finer transaction granularity and smaller conflict probability. At the end of transaction execution, the string subranges from all lazy queries are further optimized by merging intersecting subranges of all lazy queries. This optimization is done at the client side, which allows us to reduce the server load and the transaction execution time.

An *accumulated transaction* is a triple **T(I, D, V)** of strings to be inserted **I**, strings to be deleted **D**, and string intervals **V** to be verified.

Note that even if no updates were made, a transaction is still necessary to ensure the consistency of queries. Thus, a query can produce an accumulated transaction **T(I, D, V)** with empty sets **D** and **I**.

6.2. Validation Protocol

Validation is necessary to ensure two important properties of transactions: serializability and external consistency. Serializability means that the committed transactions can be ordered in such a way that the net result would be the same as if transactions ran sequentially, one at a time. External consistency means that the serialization order is not arbitrary: if transaction **S** committed before **T** began (in real time), **S** should be ordered before **T**.

When a client commits a transaction, the accumulated transaction **T** is delivered to one of the database servers. This database server is called the transaction's originator. The transaction originator splits the arriving transaction into subtransactions $\mathbf{T}_i$ according to the partitioning map and distributes the subtransactions among the database partitions. A subinterval $([l_k, r_k], t_k, p_k)$ in the set **V** is distributed to the partition p_k (without consulting the partitioning map). This allows to detect conflicts with system transactions that perform load balancing, which may change the partitioning map.

The transaction originator uses the 2-phase commit protocol to update the database. In the first phase, the transaction originator distributes the subtransactions among the database partitions. Each database partition verifies that no conflicts with any other transaction is possible and sends a "ready" or "failed" message to the transaction originator. If the transaction originator receives a "failed" message, it immediately aborts the other subtransactions and notifies the client. When all database partitions return a "ready" message, the transaction originator sends a "commit" message to the participating partitions.

In a backward validation protocol, the arriving subtransaction $\mathbf{T}_i(\mathbf{I}_i,\mathbf{D}_i,\mathbf{V}_i)$ is checked against all transactions already validated successfully. In our B-tree, each partition maintains a log of recently committed transactions **CL** and a log of transactions waiting for commit **WL**.

We say that a set of string intervals **V** intersects a set of strings **A** iff there exists an interval $[l, r]$ in **V** such that $[l,r]\ \mathbf{A} \neq \varnothing$ (i.e. for some $x \in \mathbf{A}$: $l \leq x \leq r$).

We also say that two transactions $\mathbf{T}(\mathbf{I}_T, \mathbf{D}_T, \mathbf{V}_T)$ and $\mathbf{S}(\mathbf{I}_S, \mathbf{D}_S, \mathbf{V}_S)$ intersect if:

1. $\mathbf{I}_T \cap \mathbf{D}_S \neq \varnothing$ or $\mathbf{I}_S \cap \mathbf{D}_T \neq \varnothing$
or
2. $\mathbf{V}_S$ intersects $\mathbf{I}_T \cup \mathbf{D}_T$
or
3. $\mathbf{V}_T$ intersects $\mathbf{I}_S \cup \mathbf{D}_S$

When the subtransaction $\mathbf{T}_i$ arrives, it is verified that $\mathbf{T}_i$ intersects with no transaction **S** in **WL**. Additional verification is necessary to ensure that no query in $\mathbf{T}_i$ is affected by a recently committed transaction **S** in **CL**. We check that each interval $([l_k, r_k], t_k, n_k)$ in $\mathbf{V}_i$ of $\mathbf{T}_i$ does not intersect with the sets $\mathbf{I}_S$ and $\mathbf{D}_S$ of any transaction **S** in **CL** that has greater timestamp than t_k.

If the subtransaction is successfully verified, it is appended to the **WL** and the "ready" message is sent to the transaction originator, otherwise the "failed" message is sent to the transaction originator.

While normally not requiring any disk access, this algorithm's CPU time is $\mathbf{O}(\|\mathbf{V}_i\| \times (\|\mathbf{WL}\| + \|\mathbf{CL}|\mathbf{V}_i\|))$ where $\mathbf{CL}|\mathbf{V}_i$ are the committed transactions younger than an average query in $\mathbf{V}_i$. The algorithm can be significantly accelerated by merging small sets in **CL** and **WL**.

7. Proof of Correctness

Our concurrency control algorithm satisfies both serializability and external consistency requirements. Consider two arbitrary transactions **T** and **S**. Any two subtransactions $\mathbf{T}_i$ and $\mathbf{S}_k$ of **T** and **S** for different partitions in our B-tree, are disjoint, i.e. $\mathbf{T}_i$ does not intersect $\mathbf{S}_k$ for all $\mathbf{i} \neq \mathbf{k}$. Because of this and the definition of intersection, **T** and **S** intersect iff for some partition **i** $\mathbf{T}_i$ intersects $\mathbf{S}_i$.

Definition: Relation "π" between transactions that have been successfully performed in our system: $\mathbf{T}\ \pi\ \mathbf{S}$ if any of the following conditions **(a)** and/or **(b)** holds:

(a) There exists a partition i such that $\mathbf{T}_i$ intersects $\mathbf{S}_i$ and $\mathbf{T}_i$ entered $\mathbf{CL}_i$ before $\mathbf{S}_i$.

(b) The programmatic transaction that generated **S** began execution at the client after the commit of **T** had been acknowledged to the user, i.e. **T** completed.

Lemma 1. The relation π is acyclic.
Consider an arbitrary number n of transactions and assume that there is a cycle:

$$\mathbf{T}^1\ \pi\ \mathbf{T}^2\ \pi\ \ldots\ \pi\ \mathbf{T}^n\ \pi\ \mathbf{T}^1.$$

Choose an arbitrary partition j where $\mathbf{T}^1_j$ has entered $\mathbf{CL}^j$. Let t_0 be the physical moment of time of said entry. Relation $\mathbf{T}^n\ \pi\ \mathbf{T}^1$ implies one of two cases at time t_0:

1. Condition **(b)**: The transaction $\mathbf{T}^n$ is already committed.
2. Condition **(a)**: There exists a partition i such that $\mathbf{T}^n_i$ intersects $\mathbf{T}^1_i$ and $\mathbf{T}^n_i$ entered $\mathbf{CL}_i$ before $\mathbf{T}^1_i$.

In both cases there exists a partition of $\mathbf{T}^n$ in which the physical moment of time when the subtransaction of $\mathbf{T}^n$ enters **CL** is less than t_0.

In case **1**, $\mathbf{T}^n$ is already committed and all subtransactions of $\mathbf{T}^n$ are in **CL** by the time t_0. In case **2**, the fact that $\mathbf{T}^1$ enters $\mathbf{CL}_j$ at the time t_0 and the 2-phase commit protocol implies that the other subtransactions of $\mathbf{T}^1$ must be either in **WL** or **CL** at the moment t_0. Thus, the subtransaction $\mathbf{T}^1_i$ must be in **WL** or **CL** at the moment t_0. According to our backward validation protocol $\mathbf{T}^n_i$must be in **CL** at this time (otherwise if $\mathbf{T}^n_i$ is in **WL**, the intersecting subtransaction $\mathbf{T}^1_i$ would be aborted).

Repeating the same argument n times for the relations $\mathbf{T}^{n-1}\ \pi\ \mathbf{T}^n$, ..., $\mathbf{T}^2\ \pi\ \mathbf{T}^1$, we conclude that there exists a partition k of $\mathbf{T}^1$ in which $\mathbf{T}^1$ is already in $\mathbf{CL}_k$ before time t_0.

Thus, for every partition j of $\mathbf{T}^l$ there exists a partition k in which $\mathbf{T}^l_k$ enters **CL** earlier than $\mathbf{T}^l_j$ enters $\mathbf{CL}_j$. Since $\mathbf{T}^l$ has a finite number of partitions this implication is false, in contradiction to the initial assumption.

Therefore, "π" is acyclic. ■

Theorem: Our schedule is serializable and externally consistent.

Proof:
Consider an arbitrary set of transaction that have successfully completed in the system. By the previous Lemma, "π" imposes a partial order on them. According to the known theorem that "for every partial order there exists a total order that preserves the partial order", there exists a total order "$\pi\pi$" preserving "π". The order "$\pi\pi$" defines a serialization of transactions $\mathbf{T}^l,...,\mathbf{T}^m$. The series preserves external consistency because "π" does, by its definition. It remains to show that every transaction $\mathbf{T}^k$ in the series saw the database in a state after all the previous transactions and before any subsequent transactions in the series, i.e. every query $[l,r]$ in $\mathbf{V}^k$ had resulted in

(1) $$[l,r]\sum_{i<k}(\mathbf{I}^i,\mathbf{D}^i)$$

Consider an arbitrary query $([l,r], t, p)$ in $\mathbf{V}_k$. We need to prove that

(2) $$[l,r]\sum_{\substack{i:\mathbf{T}^i \text{updated} \\ \text{before } t}}(\mathbf{I}^i_p,\mathbf{D}^i_p) = [l,r]\sum_{i<k}(\mathbf{I}^i_p,\mathbf{D}^i_p)$$

The left sum is what the query $[l,r]$ actually saw at the partition p, while the right sum is what it should have seen if serializability requirement is fulfilled. From the set-theoretical definitions of queries and transactions it follows that:

$$[l,r](A+B)=[l,r]A+[l,r]B.$$

Therefore, (2) is equivalent to:

(3) $$\sum_{\substack{i:\mathbf{T}^i \text{updated} \\ \text{before } t}}[l,r](\mathbf{I}^i_p,\mathbf{D}^i_p) = \sum_{i<k}[l,r](\mathbf{I}^i_p,\mathbf{D}^i_p)$$

Assume that (3) is not true. Consider two cases:

Case 1.
The left sum in (3) has an extra addend $[l,r]$ $(\mathbf{I}^j_p,\mathbf{D}^j_p) \neq \varnothing$, $\mathbf{k < j}$, i.e. the query $[l,r]$ may have seen the results of transaction $\mathbf{T}^j$ that was serialized after $\mathbf{T}^k$. This implies that $\mathbf{T}^k$ intersects $\mathbf{T}^j$ and $\mathbf{T}^j$ entered $\mathbf{CL}_p$ before the moment t and, therefore, before $\mathbf{T}^k$.

Since the total order relation $\pi\pi$ preserves the partial order relation π, $\mathbf{T}^k \ \pi\pi \ \mathbf{T}^j$ implies **not**($\mathbf{T}^j \ \pi \ \mathbf{T}^k$). By the definition of relation π, this in turn implies that there is no partition p such that $\mathbf{T}^j$ intersects $\mathbf{T}^k$ and $\mathbf{T}^j$ entered $\mathbf{CL}_p$ before $\mathbf{T}^k$. Thus, we have a contradiction and this case can never occur.

Case 2. The left sum does not have some addend $[l,r]\ (\mathbf{I}^j_p, \mathbf{D}^j_p) \neq \varnothing$, $\mathbf{j < k}$, i.e. $\mathbf{T}^k$ did not see some updates of transaction $\mathbf{T}^j$ ordered before $\mathbf{T}^k$. Like in the previous case, $\mathbf{T}^j \ \pi\pi \ \mathbf{T}^k$ implies **not**($\mathbf{T}^k \ \pi \ \mathbf{T}^j$) and, therefore, there is no partition p such that $\mathbf{T}^k$ intersects $\mathbf{T}^j$ and $\mathbf{T}^k$ entered $\mathbf{CL}_p$ before $\mathbf{T}^j$. Since $[l,r]\ (\mathbf{I}^j_p, \mathbf{D}^j_p) \neq \mathbf{0}$, $\mathbf{T}^j$ intersects $\mathbf{T}^k$ at the partition p. Therefore, $\mathbf{T}^j$ should have entered $\mathbf{CL}_p$ before $\mathbf{T}^k$. Thus, there is a moment when $\mathbf{T}^j$ is in $\mathbf{CL}_p$ and $\mathbf{T}^k$ is in $\mathbf{WL}_p$. Since the left sum does not have the addend $(\mathbf{I}^j, \mathbf{D}^j)$, $\mathbf{T}^j$ must have entered $\mathbf{CL}_p$ after the moment t. When $\mathbf{T}^k$ is checked by the backward validation protocol at the partition p, according to our query verification algorithm, each query $([l, r]\ t, p)$ of $\mathbf{T}^k$ should be verified against all transactions in $\mathbf{CL}_p$ with timestamp greater than t. Since $[l,r](\mathbf{I}^j_p, \mathbf{D}^j_p) \neq \varnothing$ for some transaction $\mathbf{T}^j$, the verification fails and $\mathbf{T}^k$ is aborted. Thus, in our case $\mathbf{T}^k$ would have been aborted, and we have a contradiction to our assumption that $\mathbf{T}_k$ is a committed transaction.

Thus, the equation (3) is valid and our schedule of transactions is serializable. ■

8. On Load Balancing

A query or transaction execution time is determined by its slowest subquery or subtransaction. Thus, load balancing is essential in our system to equalize the data and transactional load among B-tree partitions. In our system, load balancing is performed as a series of load balancing transactions that transfer small string intervals from one partition to another [Rishe&al-96].

Load balancing transactions require a large amount resources and time to move the data. Thus, the load balancing transactions should be well coordinated in the system. Since the data movement time is large, a centralized algorithm performs well without becoming a system bottleneck. In our B-tree, a centralized load balancing module periodically collects data and load distribution statistics from all partitions and heuristically generates a data distribution policy, initiating the load balancing transactions executed at the B-tree partitions. Apart from the load balancing transactions, the load balancing module can create a new partition or delete an empty partition which is no longer referred to by any other B-tree partition. Our concurrency control algorithm maintains its safety and efficiency notwithstanding load balancing activity done according to our algorithm of [Rishe&al-96].

9. Client String Cache

The query performance can be improved even further by using a client B-tree string cache which accumulates all strings recently retrieved from all database partitions. This allows to perform many frequently executed queries in zero server accesses. The B-tree cache stores all strings retrieved from the database servers along with the timestamp attribute and the partition number for each string.

Since the attributes of an object constitute lexicographically close strings, it is likely that after requesting a string the client will later request lexicographically close strings. Thus, the cache

performance can be improved by prefetching more than one B-tree block that contain the requested query.

The client cache significantly improves the performance of our optimistic concurrency protocol. When a client transaction fails due to a conflict with another transaction which affects some queries, a list of failed (affected) queries is returned to the client. All failed string ranges are invalidated in the client cache, and when the transaction is executed again, the client retrieves the invalidated ranges from the server. When the transaction runs the second time, most of the other ranges that the user retrieves are already in the string cache and the number of server accesses is much smaller. This can significantly improve the transaction execution time at the second attempt. Since the conflict probability is proportional to the execution time, the conflict probability is also reduced.

10. Livelocks and Exceptional Pessimistic Locks

In rare situations, a transaction may be repeatedly aborted and reexecuted, causing a livelock. To avoid this, we introduce a pessimistic element in our concurrency control. When a privileged user desires to protect a new transaction, the user can also designate it as pessimistic.

A transaction in the pessimistic mode has a unique logical identifier assigned by the system. This identifier has smaller value for older transactions or transactions with higher priority. A transaction executed in the optimistic mode is "assigned" an infinitely large logical identifier.

A transactional query in a pessimistic mode does not use cache, but is sent directly to the corresponding database partitions. Each partition maintains an additional log called **PL** which logs all pessimistic transactional queries as soon as they are executed. Whenever a new transaction **T** conflicts with a query in **PL** that has a smaller logical identifier than **T**, **T** is aborted. To ensure that a query in pessimistic transaction does not conflict with any transaction which is ready to commit, the query execution should be delayed until there is no transaction in **WL** that conflicts with the query. When an accumulated pessimistic transaction **T(I,D,V)** arrives, its verification should be delayed until there is no transaction in **WL** that conflict with the sets **D** or **I**. A query is removed from **PL** when its transaction arrives for verification or after a timeout period, whichever happens first.

This algorithm guarantees successful execution of the transaction with the smallest logical identifier.

11. Conclusion

In this paper we described our efficient concurrency control algorithm that we used in implementation of a parallel B-tree server. This algorithm has very high granularity while avoiding high storage and processing time overheads. Our algorithm uses logical clocks and does not require physical clock synchronization. The B-tree server can handle variable length keys and can be used in a variety of databases, including relational, object-oriented, and semantic. Many of our B-tree features (data compression, concurrency control, lazy queries) have been implemented in C++ and tested. Preliminary results demonstrated very good performance.

Lazy queries in a B-tree can significantly improve server performance in case of complex and large queries. Lazy queries can also decrease the transaction conflict probability, which is

essential for on-line transaction processing systems where high contention workloads are common. Our algorithm has a very fine granularity (attribute or string level granularity), which also contributes to smaller transaction conflict probability.

12. References

[Adya&al-95] A. Adya, R. Gruber, B. Liskov, U. Masheshwari. "Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks," *SIGMOD Record*, June 1995, v 24 n. 2, pp. 23-34.

[Comer-79] D. Comer. "The Ubiquitous B-tree," *ACM Computing Surveys*, June 1979, v 11 n. 2.

[Gray-79] J. Gray. "Notes on Database Operating Systems" in R. Bayer, R. Graham, and G. Seegmuller. *Operating Systems: An Advanced Course*, pp. 394-481, Spring-Verlag, 1979.

[Haerder-84] T. Haerder. "Observations on Optimistic Concurrency Control," *Information Systems*, June 1984, v 9 n. 2 , pp. 111-120.

[Rishe&al-96] N. Rishe, A. Shaposhnikov, S. Graham. "Load Balancing in a massively parallel semantic database" to appear in the International Journal of Computer Science and Engineering.

[Rishe-92-DDS] N. Rishe. *Database Design: The Semantic Modeling Approach.* McGraw-Hill, 1992, 528 pp.

[Rishe-91-FS] N. Rishe. "A File Structure for Semantic Databases", *Information Systems*, v 16 n. 4, 1991, pp. 375-385.

[Rishe-91-IB] N. Rishe. "Interval-based approach to lexicographic representation and compression of numeric data," *Data & Knowledge Engineering*, n 8, 1992, pp. 339-351.

Comput Syst Sci & Eng (1996) 4: 195–199

Computer Systems
Science & Engineering

Load balancing in a massively parallel semantic database

Naphtali Rishie, Artyom Shaposhnikov and Scott Graham

School of Computer Science, Florida International University, University Park, Miami, FL 33199, USA. Email: rishen@fiu.edu

We are developing a massively parallel semantic database machine. Our basic semantic storage structure ensures balanced load for most parts of the database. The load to the other parts of the database is kept balanced by a heuristic algorithm which repartitions data among processors in our database machine as necessary to produce a more evenly balanced load. We present our inexpensive, dynamic load balancing method together with a fault- tolerant data transfer policy that will be used to transfer the repartitioned data in a way transparent to the users of the database.

Keywords: DBMS, massive parallelism, semantic data models, load balancing, database machine

1. INTRODUCTION

Database management systems are emerging as prime targets for enhancement through parallelism. In order for parallel database machines to be efficient, the processors in the system must have comparable load. A massively parallel database machine which uses thousands of processors will allow for massive throughput of transactions and queries if no processors become a bottleneck. This paper proposes a load balancing method for a massively parallel semantic database.

Much work on load balancing for relational databases and file systems has been done and can be utilized in our research. For example, Sitaram *et al.*[1] propose several dynamic load balancing policies for multi-server file systems. A dynamic load balancing algorithm for large, shared-nothing, hypercube database computers which makes use of relational join strategies is presented in Hua and Su[2]. Lee and Hua[3] present a self-adjusting data distribution scheme which balances computer workload in a multiprocessor database system at a cell level during query processing. A run-time reorganization scheme for rule based processing in large databases is discussed in Stolfo *et al.*[4].

Our database computer will make use of a shared-nothing architecture. The computational load on each processor of our database computer will vary directly with the demand for data on that processor. Imbalances in the number of data accesses among nodes can be rectified by repartitioning the database, much as imbalances in computational demands in process scheduling can be rectified by moving processes from one machine to another. When a range of facts in our database is moved from one processor's control to another processor's control, the load on the first processor will go down. The methods for determining imbalances in our system, and the methods to relieve these imbalances in our system, are very similar to the methods used for computational dynamic load balancing in shared-nothing computers. An adaptive, heuristic method for dynamic load balancing in a message-passing multicomputer is presented in Xu and Hwang[5]. A semi-distributed approach to load balancing in massively parallel multicomputer systems is presented in Ahmad and Ghafoor[6].

Our massively parallel database machine architecture makes use of a distributed system of many processors, each with its own permanent storage device. This shared-nothing approach requires that any load balancing operations be performed by message passing. The data distribution scheme that is used in our database system allows load balancing to be achieved by

data repartitioning among the nodes of our system.

This paper refines the results reported in Rishie *et al.*[7] and extends them by adding a fault tolerant data transfer policy for data repartitioning.

2. SEMANTIC BINARY DATABASE MODEL

The semantic database models in general, and the Semantic Binary Model SBM (Rishie[8] and others) in particular, represent the information of an application's world as a collection of elementary facts categorizing objects or establishing relationships of various kinds between pairs of objects. The central notion of semantic models is the concept of an *abstract object*, which is any real world entity that we wish to store information about in the database. The objects are categorized into classes according to their common properties. These classes, called *categories*, need not be disjoint – that is, one object may belong to several classes. Further, an arbitrary structure of sub-categories and super-categories can be defined. The representation of the objects in the computer is invisible to the user, who perceives the objects as real-world entities, whether tangible, such as persons or cars, or intangible, such as observations, meetings, or desires. The database is perceived by its user as a set of facts about objects. These facts are of three types: facts stating that an object belongs to a category: *xC*; facts stating that there is a relationship between objects: *xRy*; and facts relating objects to data, such as numbers, texts, dates, images, tabulated or analytical functions, etc: *xRv*. The relationships can be of arbitrary kinds; stating, for example, that there is a many-to-many relation *address* between the category of persons and texts means that one person may have an address, several addresses, or no address at all.

3. STORAGE STRUCTURE

An efficient storage structure for semantic models has been proposed in Rishie[9, 10]. The collection of facts forming the database is represented by a file structure which ensures approximately 1 disk access to retrieve queries of any of the following forms:

1. For a given abstract object x, verify/find what categories the object belongs to.
2. For a given category, find its objects.
3. For a given abstract object x and relation R, retrieve all/certain y such that xRy.
4. For a given abstract object y and relation R, retrieve all/certain abstract objects x such that xRy.
5. For a given abstract object x, retrieve (in one access) all (or several) of its direct and/or inverse relationships, i.e. relations R and objects y such that xRy or yRx. The relation *R* in xRy may be an attribute, i.e. a relation between abstract objects and concrete objects.
6. For a given relation (attribute) R and a given concrete object y, find all abstract objects such that xRy.
7. For a given relation (attribute) R and a given range of concrete objects $[y_1, y_2]$, find all objects x and y such that xRy and $y \leq y_1 \leq y_2$.

The entire database can be stored in a single file. This file contains all of the facts of the database (*xC* and *xRy*) as well as additional information called inverted facts: *Cx*, *Ryx*. The inverted facts ensure that answers to queries of forms 2, 4, 6 and 7 are kept in a contiguous segment of data in the database which allows them to be answered with one disk access. The direct facts *xC* and *xRy* allow the database to answer queries of forms 1, 3, and 5 with one disk access. The file is maintained as a B-tree. The variation of the B-tree used allows both sequential access according to the lexicographic order of the items comprising the facts and the inverted facts, as well as random access by arbitrary prefixes of such facts and inverted facts. Facts which are close to each other in the lexicographic order reside close to each other in the file. (Notice that although technically the B-tree-key is the entire fact, it is of varying length and on the average is only several bytes long, which is the average size of the encoded fact *xRy*.)

Consider, for example, a database containing information regarding products manufactured by different companies. The following set of facts can be a part of a logical instantaneous database:

1. object1 ***COMPANY***
2. object1 ***COMPANY-NAME*** 'IBM'
3. object1 ***MANUFACTURED*** object2
4. object1 ***MANUFACTURED*** object3
5. object2 ***PRODUCT***
6. object2 ***COST*** 3600
7. object2 ***DESCRIPTION*** 'Thinkpad'
8. object3 ***PRODUCT***
9. object3 ***COST*** 100
10. object3 ***DESCRIPTION*** 'TrackPoint'

The additional inverted facts stored in the database are:

1. ***COMPANY*** object1
2. ***COMPANY-NAME*** 'IBM' object1
3. object2 ***MANUFACTURED-BY*** object1
4. object3 ***MANUFACTURED-BY*** object1
5. ***COST*** 3600 object2
6. ***COST*** 100 object3
7. ***DESCRIPTION*** 'Thinkpad' object2
8. ***DESCRIPTION*** 'TrackPoint' object3
9. ***PRODUCT*** object2
10. ***PRODUCT*** object3

To answer the elementary query "Find all objects manufactured by object1", we find all the facts whose prefix is object1_***MANUFACTURED***. ('_' denotes concatenation.) These entries are clustered together in the sorted order of direct facts.

To answer the elementary query "Find all products costing between $0 and $800", we find all the facts whose prefix is in the range from ***COST***_0 to ***COST***_800. These entries are clustered together in the sorted order of inverted facts.

In the massively parallel version that we are developing,

the B-tree is pa
residing on a sep
memory) that is
This disk-proces
retrieve informat
processing on the
the other nodes.
vant integrity co
mation on the dis
or updated concu

The queries a
through host int
copy of the Part
Since the whole
represented by a
only a small num
cally minimal ar
that is stored on
database is re-p
propose in this
inexpensive, loc
the shifting of d
with the normal

Most of the pl
of a semantic bi
These facts are
objects, which a
each abstract obj
and since the obj
that traffic to ea
over time. Other
with an inverted
tion between an
possible that at
certain attribute
or categories. Th
values of a given
ticular inverted
together, this
processor/disk pa
can occur in som
taining the facts
file will contain
object. The secon
with an inverted
third subfile cont
which are pointe
partitioned even
system. The first
third subfiles ma
block placement
tioned. By reparti
ly balance the loa

4. REQUI

We employ a de
cessing. This mea
formed until they
database managen

the B-tree is partitioned into many small fragments, each residing on a separate storage unit (e.g. a disk or non-volatile memory) that is associated with a fairly powerful processor. This disk-processor pair is called a *node*. Each node can retrieve information from the disk, perform the necessary processing on the data and deliver the result to the user, or to the other nodes. For updates the node verifies all of the relevant integrity constraints and then stores the updated information on the disk. Many database fragments can be queried or updated concurrently.

The queries and transactions will enter into the network through host interfaces. Every host interface maintains a copy of the Partitioning Map (PM) of the entire database. Since the whole database is a lexicographically ordered file represented by a set of B-trees, the map needs to contain only a small number of facts for each node: the lexicographically minimal and maximal facts for each B-tree fragment that is stored on that node. The map changes only when the database is re-partitioned. The distribution policy that we propose in this work provides repartitioning that is rare, inexpensive, localizable, invisible to the system until all of the shifting of data is complete, and that does not interfere with the normal operation of the system.

Most of the physical facts that are in our implementation of a semantic binary database start with an abstract object. These facts are ordered by the encoding of the abstract objects, which assigns a unique quasi-random number to each abstract object. Since there are so many of these facts, and since the objects are randomly ordered, we can assume that traffic to each partition of these facts will be balanced over time. Other facts in a semantic binary database start with an inverted category or an inverted attribute (i.e. a relation between an abstract object and a printable value). It is possible that at some time there may be a need to access a certain attribute or category more often than other attributes or categories. The same may be true for a specific range of values of a given attribute. Since all facts that refer to a particular inverted attribute or inverted category are clustered together, this may cause a higher load on some processor/disk pairs than on others. Since load imbalances can occur in some kinds of facts but not others, the file containing the facts will be split into two subfiles. The first subfile will contain all the facts that begin with an abstract object. The second subfile will contain the facts that begin with an inverted attribute or category. Additionally there is a third subfile containing long data items: texts, images, etc., which are pointed to by facts. Each subfile will be initially partitioned evenly over all the processor/disk pairs in the system. The first subfile is already balanced; the second and third subfiles may become unbalanced and will require a block placement algorithm that allows the data to be repartitioned. By repartitioning data, we will be able to more evenly balance the load to each data partition.

4. REQUEST EXECUTION SCHEME

We employ a deferred update scheme for transaction processing. This means that transactions are not physically performed until they are committed, but are accumulated by the database management system as they are run. Upon completion of the transaction the DBMS checks its integrity and then physically performs the update. A completed transaction is composed of a set of facts to be deleted from the database, a set of facts to be inserted into the database, and additional information needed to verify that there is no interference between transactions of concurrent programs. In our parallel database, each node is responsible for a portion of the database. When an accumulated transaction is performed, the sets of facts to be inserted into, and deleted from, the database must be broken down into subsets that can be sent to the processors which are responsible for the relevant ranges of data.

Each host in the system will have a copy of the Partitioning Map (PM). The Partitioning Map is a small semantic database containing information about data distribution in the system. Figure 1 is a semantic schema of the partitioning map.

The partitioning map contains a set of ranges and their lexicographical bounds – the *low-bound* and the *high-bound* values. When a query or transaction arrives, the host will identify its lexicographical bounds. The host will then use the partitioning map to determine a set of ranges that needs to be retrieved or updated and hence the nodes which will be involved in the current transaction or query.

The partitioning map will be replicated among hosts. However, this does not imply that we need a global data structure; the algorithm described below allows updates of the partitioning database to be performed gradually, without locking and interrupting all hosts.

A range can be obtained from the node pointed to by the *location* reference in the partitioning database. This node should either have the range or a reference to another node which contains the range.

To perform load balancing we will need to move ranges from one node to another. A moved range will be accessible via an indirect reference that is left at its previous location. Such an indirect access slows down the system, especially when the range is frequently accessed by users. To allow a direct access to the moved range we need to update the *location* reference in the partitioning database. We will not perform this update simultaneously for all the host interfaces. The update will be performed when a host executes the first query or transaction that refers to the range that was transferred. The node that actually holds the range will send the results to the host along with a request to update the partitioning map. This means that the first transaction will have to travel a little further than all subsequent transactions. The second and future queries or transactions made through this host will be executed directly by the node pointed to by the *location* reference.

The data structure at each node which supports indirect referencing will be exactly the same as the partitioning map described above. We will call this data structure a local partitioning map.

Each range of facts will be represented as a separate B-tree structure which will reside on the node pointed to by the partitioning map. Consider a case where a range has been moved several times from one node to another. We may have multiple indirection references to the actual location of the range. These indirect references will be changed to direct references as described above.

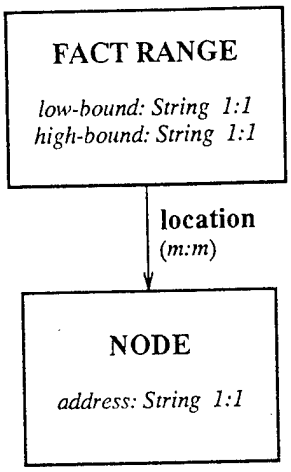


Figure 1 Partitioning map

5. DATA TRANSFER POLICY

In order to ensure that the database remains consistent throughout a load balancing data transfer, load balancing actions are executed as transactions initiated by the system. A large range of facts is transferred by executing a series of small system transactions that transfer small portions of data from one partition to another. The system transactions are subject to the same logging and recovery actions as regular, user initiated, transactions. Apart from the data transfer, each small load balancing transaction also includes the data necessary for updating the partitioning map. To ensure that the partitioning map remains consistent, the partitioning map update is executed using a 2-phase commit protocol.

6. LOAD BALANCING POLICY

When idle, the host interfaces will send data and work load statistics recently accumulated from the nodes to a Global Performance Analyzer (GPA). The host interfaces accumulate this data as the results of queries and transactions flow through them back to the user. The GPA is a process that analyzes the statistical information obtained and generates preferable directions of data transfer for each node.

The statistics report will contain only the changes since the previous report:

- Changes in data partitioning
- Number of accesses for each range
- Free space on each node

The GPA will use a heuristic search algorithm which uses a choice function to select a small number of possible data movements for the system. The final state will be estimated by a static evaluation function S. The GPA will select the data movement with the lowest value of the resulting static evaluation S.

The choice function should comply with the following strategies:

1. Whenever possible load balancing should be achieved by joining ranges together. Joining ranges will result in faster query execution and smaller partitioning maps.
2. A criterion for determining preferable destinations for a range transfer is the desire to move a range to a destination node which contains the lexicographically closest range to the transferred range. In other words, it is desirable to locate lexicographically close ranges on the same node whenever possible.
3. If a range has an exceptionally high number of access or requires an exceptionally large amount of storage – split the range into several parts and transfer them to other nodes.

Each node will be characterized by two parameters:

1. The amount of free disk space on the node. F
2. The percentage of idle time I. In other words the I is: $I = Idle/T$, where T is a given time interval and $Idle$ is the node's idle time during the time T.

The resulting state will be estimated by the following parameters:

1. A – the total amount of data that will be necessary to transfer in the system
2. D_F – the mean square deviation of F
3. D_I – the mean square deviation of I
4. M – total number of ranges in the system

The static evaluation function can be represented as:

$$S = C_1 * A + C_2 * D_F + C_3 * D_I + C_4 * M,$$

where C_1, C_2, C_3 and C_4 are constants.

7. CONCLUSION

Our load balancing algorithm will provide our massively parallel semantic database machine with a method to repartition data to evenly distribute work among its processors. The algorithm has very little overhead, as its statistics are accumulated during the normal processing of transactions. The load balancing is accomplished by repartitioning parts of the database over the nodes of the database machine. The repartitioning will be transparent to the users and will not adversely affect the performance of the system. Our fault-tolerant data transfer policy will ensure that the database and its partitioning maps remain consistent during repartitioning.

We are currently developing a prototype parallel semantic database on a network of workstations. We will evaluate our load balancing algorithm on this prototype system and experiment with ways to optimize our heuristic search algorithm.

ACKNOWLEDGEMENTS

This research was supported in part by NASA (under grant NAGW-4080)
NATO (under
CDA-931362-

REFEREN

1 **Sitara** Multi- *Proce Paral* Diego Press.
2 **Hua.** Large *IEEE* her 1
3 **Lee.** Mech. Multi *Syste*
4 **Stolfo** Parali Proce *Emer*

NAGW-4080), BMDO&ARO (under grant DAAH04-0024), NATO (under grant HTECH.LG-931449), NSF (under grant CDA-9313624 for CATE Lab), and the State of Florida.

REFERENCES

1 **Sitaram, D, Dan, A and Yu, P** 'Issues in the Design of Multi-Server File Systems to Cope with Load Skew', *Proceedings of the Second International Conference on Parallel and Distributed Information Systems* (San Diego, January 20-22, 1993), IEEE Computer Society Press, 1993

2 **Hua, K and Su, J** 'Dynamic Load Balancing in Very Large Shared-Nothing Hypercube Database Computers', *IEEE Transactions on Computers*, Vol 42 No 12 (December 1993) pp 1425-1439

3 **Lee, C and Hua, K** 'A Self-Adjusting Data Distribution Mechanism for Multidimensional Load Balancing in Multiprocessor-Based Database Systems', *Information Systems*, Vol 18 No 7 (1994) pp 549-567

4 **Stolfo, S, Dewan, H, Ohsie, D and Hernandez, M** 'A Parallel and Distributed Environment for Database Rule Processing: Open Problems and Future Directions', in *Emerging Trends in Database and Knowledge-Base Machines: The Application of Parallel Architectures to Smart Information Systems*, M. Abdelguerfi and S. Lavington, eds, IEEE Computer Society Press, 1995, pp 225-253

5 **Xu, J and Hwang, K** 'Heuristic Methods for Dynamic Load Balancing in a Message-Passing Multicomputer', *Journal of Parallel and Distributed Computing*, Vol 18 (1993) pp 1-13

6 **Ahmad, I and Ghafoor, A** 'Semi-Distributed Load Balancing For Massively Parallel Multicomputer Systems', *IEEE Transactions on Software Engineering*, Vol 17 No 10 (October 1991) pp 987-1004

7 **Rishe, N, Shaposhnikov, A and Sun, W** 'Load Balancing Policy in a Massively Parallel Semantic Database', *Proceedings of the First International Conference on Massively Parallel Computing Systems*, IEEE Computer Society Press, 1994, pp 328-333

8 **Rishe, N** *Database Design: The Semantic Modeling Approach*, McGraw-Hill, 1992

9 **Rishe, N** 'Efficient Organization of Semantic Databases', *Foundations of Data Organization and Algorithms (FODO-89)* W. Litwin and H.-J. Schek, eds., Springer-Verlag *Lecture Notes in Computer Science, Vol 367*, pp 114-127, 1989

10 **Rishe, N** 'A File Structure for Semantic Databases', *Information Systems*, Vol 16 No 4 (1991) pp 375-385

SB2 Benchmark (Consumer Survey Database)

The purpose of the test database is to store the information gathered in a typical consumer survey. All consumers are grouped by the type of the product they use. These groups are represented in the schema by subcategories G0,G1,...G9. A consumer usually belongs to several groups. Within each group a consumer may use several brands of the same product. The integer attributes A0,A1,...A9 are used to indicate which brands the consumer uses in the order of preference. Sometimes a consumer may want to enter a comment about any brand he uses. Comments for the corresponding brands are stored in the attributes C0,C1,...C9. Comments are entered very rarely.

The problem domain can be easily represented in a semantic schema. A relational schema for Oracle allows several different designs. The design choice impacts both the space requirements for the database, and the efficiency of the transactions performed. Designing a relational schema we must take into consideration the characteristics of the data that we intend to store (which is not required for the semantic schema design). Since the tables G1,G2,...G9 are going to be sparse, we have two reasonable choices for the schema design, and we have implemented both.

In the first design, which we call "Sparse," groups are represented as different tables like in the semantic schema. In the second design, which we call "Compact," all the data for all the groups is stored in one table having a three-attribute key (consumer, group and brand). This Compact approach is intended to save space, as it contains rows only for those brands that are actually used by the corresponding consumer.

The database must be able to efficiently respond to arbitrary queries. Therefore, the relational database was created fully indexed. Further, Oracle was allowed to gather statistics on the database prior to running the benchmark transactions.

In the Compact relational design the database itself required less space than in Sparse relational design, but the total occupied space including indexes was comparable for both designs, and about 3 times the space required by Semantic Database. The benchmark tests were performed for two different database sizes. In the first test, the initial database contained 100,000 consumers, in the second - 500,000 consumers. The actual database sizes are shown in the results section.

Oracle transactions were written using the Embedded SQL. Transactions for the Semantic Database were written using semantic API. The same row data files were loaded into the Oracle and the Semantic Database.

The initial data for the SB2 Benchmark Database (Consumer Survey Database)

The consumer's name, address and comment are strings of random alpha characters. The length of the string is generated as a random number with a Normal Distribution (a table of distribution parameters is given).

Each consumer can belong to a number of groups and within that group he can use a number of brands of the product. He can have a number of hobbies and use a number of stores. All these numbers are random with a Normal Distribution and parameters according to the table.

Comment fields are filled for 2% of corresponding numeric brand preference values.

SSN is a random number uniformly distributed in the range 100,000,000 .. 999,999,999.

For the purpose of establishing of a "knows" relation all consumers are divided into disjoint sets $S_1..S_n$, where each set contains exactly 10 consumers. Then 5 random consumers from each set S_i are related to 5 random consumers from the set S_{i+1}.

There are 20 different stores with names "Store name #1" through "Store name #20" and types "Store type #1" through "Store type #3".

Expenditure is a random number with a Normal Distribution and parameters according to the table.

Cid is the ID of a consumer, assigned sequentially.

Table of Normal Distribution parameters:

	Lower bound	Upper bound	Mean	Variance
Name length	5	40	12	5
Address length	15	100	35	20
Comment length	5	255	30	100
Number of hobbies per consumer	0	19	0	10
Number of stores per consumer	1	19	4	10
Expenditure	1	89	20	10
Number of groups a consumer belongs to	1	10	5	4
Number of brands a consumer uses	0	9	1	1

SB2 Benchmark (Consumer Survey Database)
Transactions

Transaction 1:

How many consumers are in the intersection of the ten groups G0..G9.

Transaction 2:

Create a new group G10 and populate it with those consumers who belong to both G1 and G2 and have A1=1 in G1 and A2=2 in G2.

Transaction 3:

How many consumers are customers of store X and have hobby Y, excluding those who belong to both G3 and G4 and have A3=3 in G3 and A4=4 in G4.

Transaction 4:

For each person from a given (randomly chosen) set of 0.1% of all consumers, expand the relation "knows" to relate this person to all people he has a chain of acquaintance to. Abort the transaction rather than commit. Print the length of the maximal chain from the person.

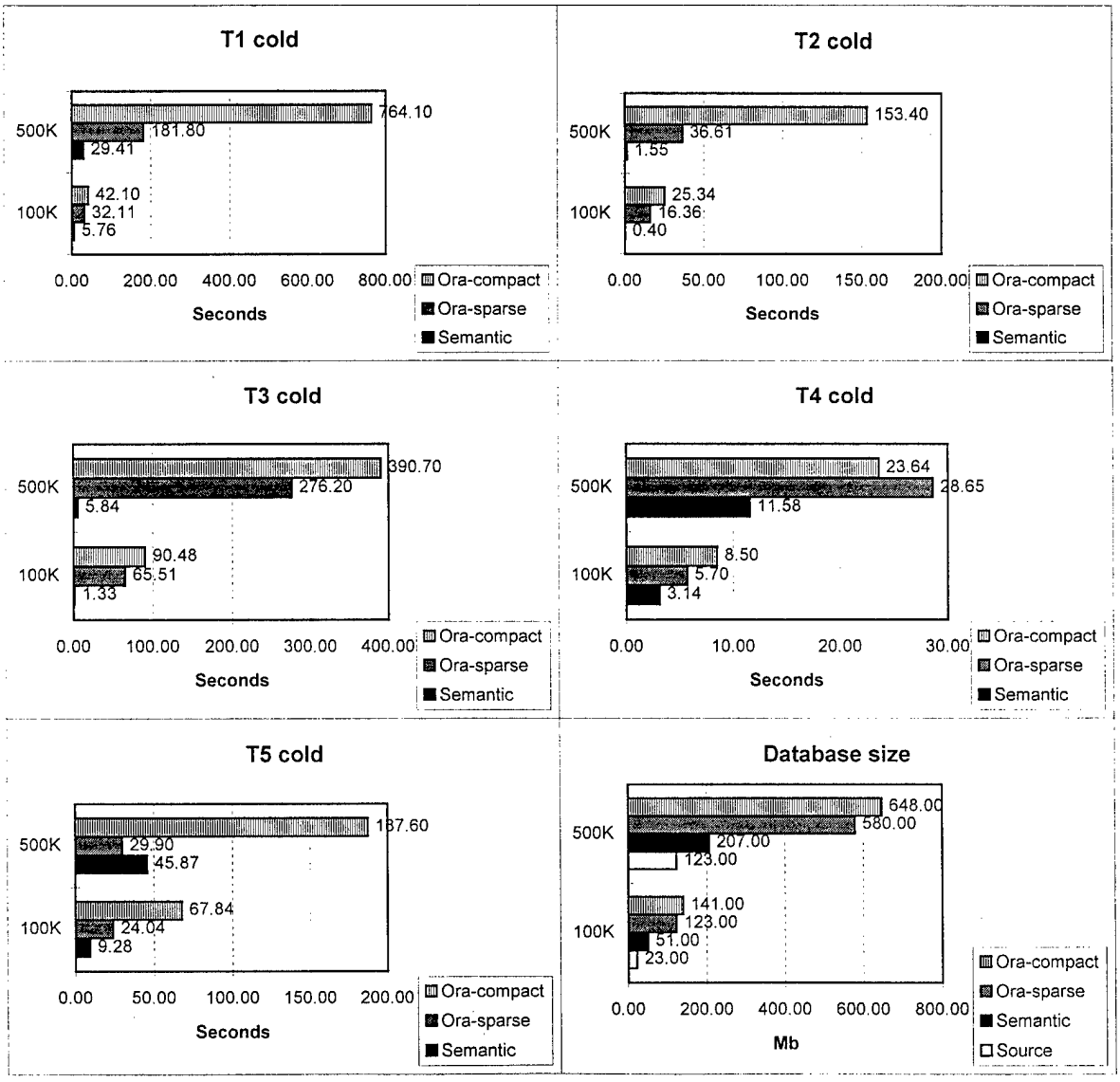
Transaction 5:

Calculate the number of consumers in each group.

SB2 Benchmark (Consumer Survey Database) : SDB vs Oracle

SB2

Database	Semantic		Oracle Compact		Oracle Sparse	
# of consumers	100K	500K	100K	500K	100K	500K
Source size (Mb)	23.00	123.00	23.00	123.00	23.00	123.00
Database size (Mb)	51.00	207.00	141.00	648.00	123.00	580.00
T1 cold (sec)	5.76	29.41	42.10	764.10	32.11	181.80
T1 hot (sec)	4.95	25.62	28.82	727.95	4.21	21.65
T2 cold (sec)	0.40	1.55	25.34	153.40	16.36	36.61
T2 hot (sec)	0.11	0.29	2.75	13.88	0.84	1.20
T3 cold (sec)	1.33	5.84	90.48	390.70	65.51	276.20
T3 hot (sec)	1.16	5.55	48.34	389.63	45.28	223.28
T4 cold (sec)	3.14	11.58	8.50	23.64	5.70	28.65
T4 hot (sec)	0.08	0.28	1.18	10.22	1.11	7.43
T5 cold (sec)	9.28	45.87	67.84	187.60	24.04	29.90
T5 hot (sec)	8.70	43.86	36.86	187.83	5.62	28.05



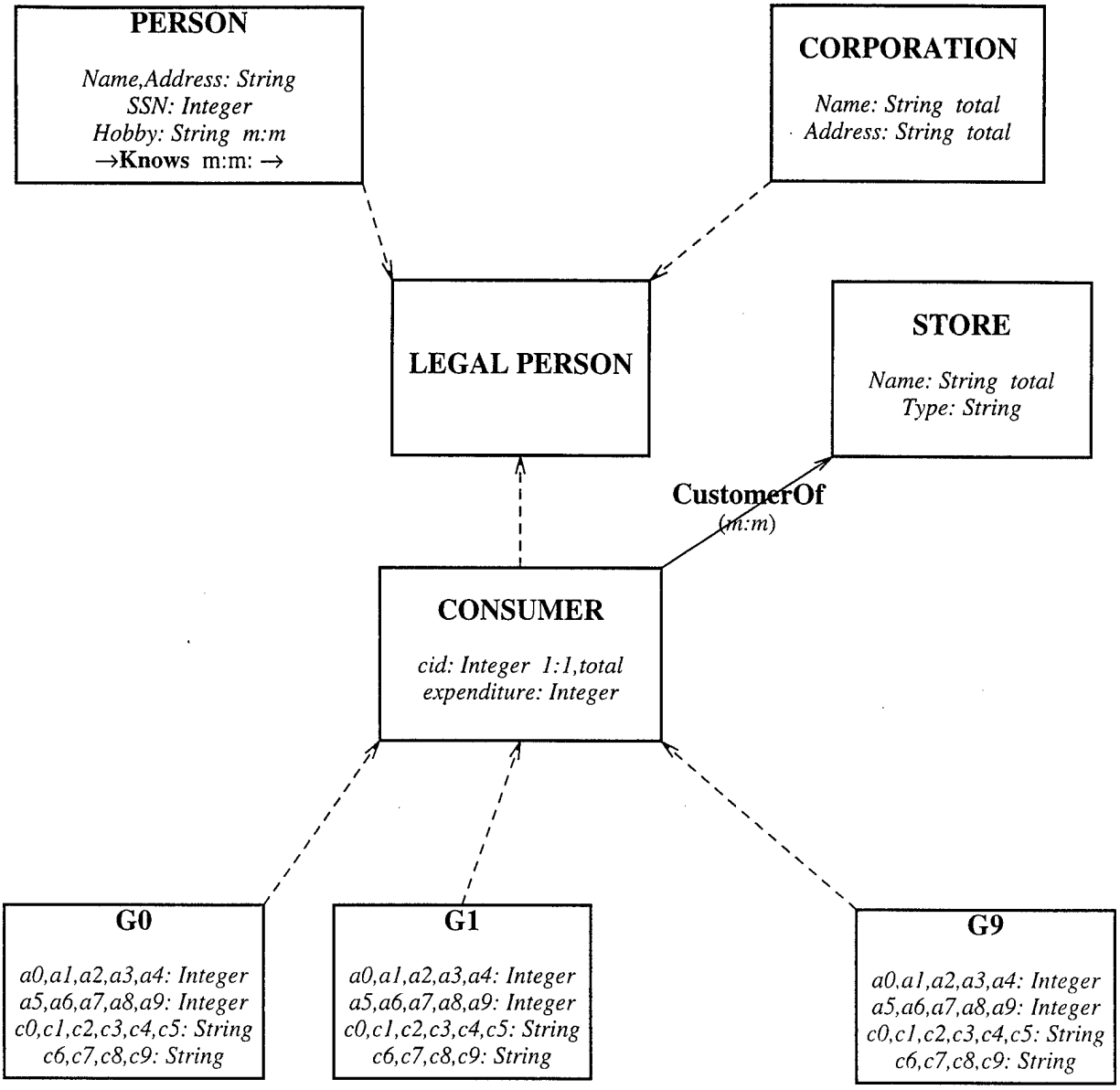


Figure 1. Semantic Schema for SB2 Benchmark

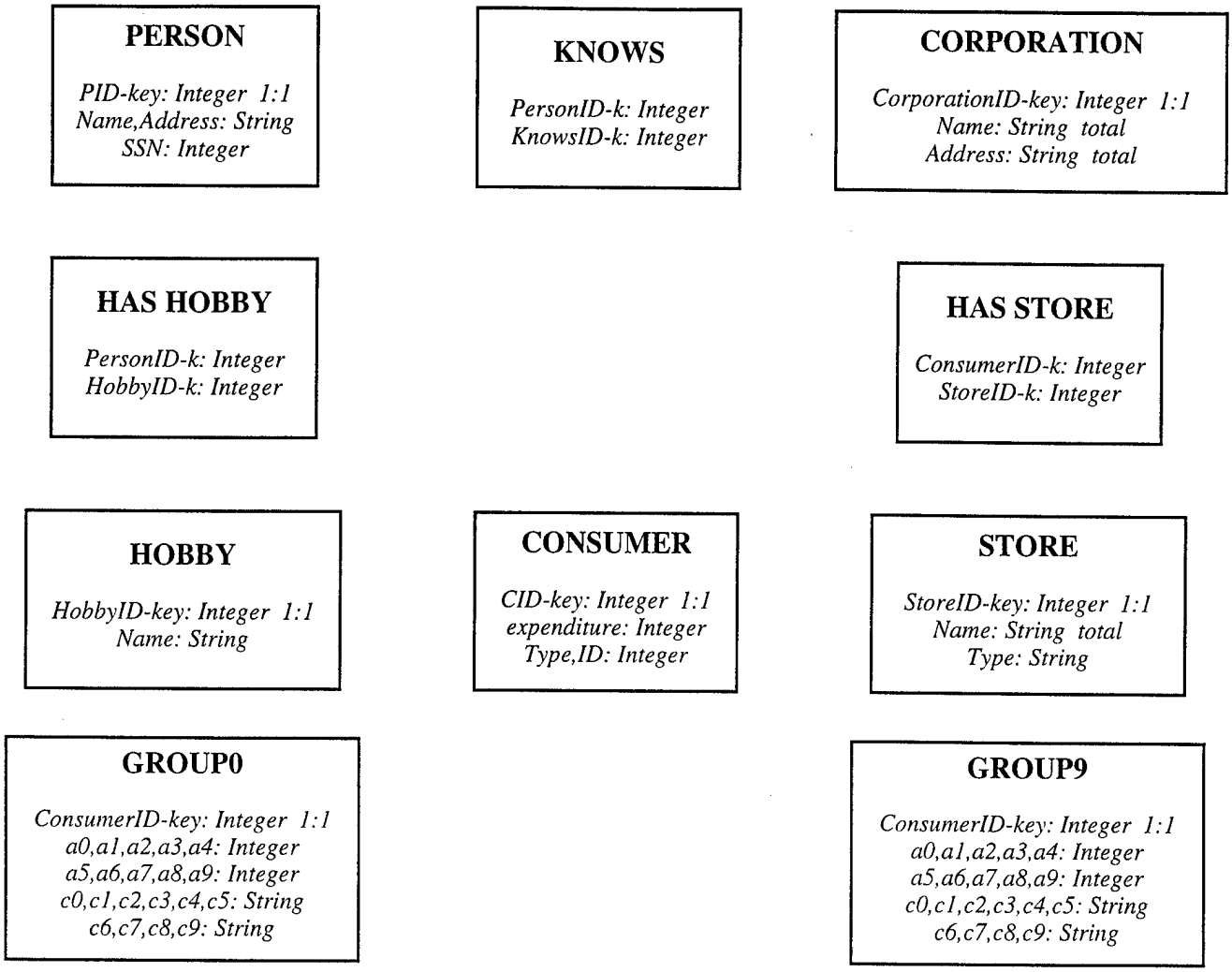


Figure 1. Relational Schema for SB2 Benchmark (Sparse alternative)

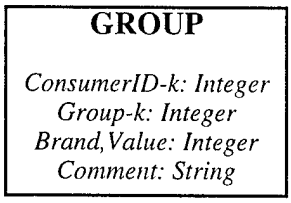


Figure 2. Relational Schema for SB2 Benchmark (Compact alternative)

Oracle is fully indexed. Best results are chosen for Oracle with/without statistics.

Semantic SQL

Naphtali Rishe

High-performance Database Research Center
School of Computer Science
Florida International University, University Park, Miami, FL 33199
(305)348-2025, Fax (305)348-1707, rishen@fiu.edu, http://hpdrc.cs.fiu.edu

We have adapted SQL, the standard relational database language, to semantic databases. The original purpose of this adaptation was to be compatible with, and be able to communicate with, relational tools. Interestingly, it turned out that the size of a typical SQL program for a semantic database is many times smaller than for an equivalent relational database. While we have previously demonstrated substantial program-size advantage for other languages, we had not anticipated an even greater advantage with SQL — a specialized language for relational databases.

Our ODBC driver for the SDB Engine is fully operational, allowing SQL querying of a semantic database and interoperability with relational database tools, e.g. end-user systems like MS Access Query-By-Example. In these tools the number of user keystrokes required is proportional to the size of the generated SQL program. So again, savings are realized and simplicity is attained by use of the SDB model.

An embedded SQL preprocessor has been developed and is fully operational.

Our application of SQL to semantic databases allows utilization of full semantics of data, applies to scientific and spatial data, properly treats missing values, and produces queries which are typically an order of magnitude shorter than if written in SQL for an equivalent normalized relational database — see examples in Section 4.

1. SQL INTERPRETATION

We use the same syntax as the standard ODBC SQL (with null values). However, our SQL queries refer to a virtual schema. This virtual schema consists of an inferred table T defined for each category C as a spanning tree of all the relations reachable from C. This is recursively defined as follows:

Let C be a category.

(1) The first attribute of T:

- ☐ *C* — attribute of *T*, range: *C* (*m:1*)

(2) For every attribute A of T, for every relation *r* whose domain intersects with the range of A:

☐ *A__r* — attribute of *T*, range: *range(r)* *(m:1)*
provided the depth of recursion does not exceed the system variable $MAXDEPTH

If the original relation *r* is many-to-many or one-to-many, the new attribute would be many-to-one, but many virtual rows would exist in the table T, one for each instance of the tree. If *r* has no value for an object, a null value will appear in the virtual relational table.

The name of T is the same as of C.

The attribute names of T contain long prefixes. These prefixes can be omitted when no ambiguity arises, i.e.. attribute y is a synonym of the attribute x_y of T if T has no other attribute z_y where depth(z)>=depth(x).

We note that the range of a virtual attribute may be of multi-media type: numbers with unlimited varying precision and magnitude, texts of unlimited size, images, etc.

Prior to computing the virtual tables, we eliminate all special characters, including underscores, from concept names; we augment the schema or the user-view with the following virtual relations:

- inverted relations: for every relation R, its inverse is called, by default, _R
- for every category C, a surrogate attribute, also called C. This is the identity attribute on C. It can be used for checking on belonging to a subcategory (let p be a PERSON.; p is a student iff p.STUDENT is not null) or to produce a printable id of an object (see Appendix).
- for every category, a combined attribute C__, which is the concatenation of all attributes of C that are representable by printable strings (this includes numbers, enumerated, Boolean. The concatenated values are separated by slashes. Null values are replaced by empty strings.
- infinite virtual relations representing functions over space-time, which in the actual database are represented by a finite data structure.

2. TECHNICAL NOTES

2.1. Definition of the Extension of a Table

The virtual table T for a category C is logically generated as follows:

(1) Initially, T[C]=C, i.e. T contains one column called C whose values are the objects of the category.

(2) For every attribute A of T, for every schema relation or attribute r whose domain may intersect range(A), let R be the relation r with its domain renamed A and range renamed A__r, let T be the natural right-outer-join of T with R. (Unlike a regular join, the outer join creates A__r=null when there is no match.)

2.2. Surrogate Attributes

In Release 0.1, the surrogate attribute of each semantic category is the internal id of the object.

In Release 0.2, the surrogate attribute will be defined in accordance with our document on surrogates (see Appendix).

2.3. User-control of Table Depth

(Used only by sophisticated users trying to outsmart $MAXDEPTH defined by a graphical user interface; not needed by users posing direct SQL queries without a GUI.)

For each category C, in addition to the default table named C, of depth limited by $MAXDEPTH, there are also tables called C_i for any positive integer i, with the depth limited by i rather than $MAXDEPTH. The tables C_i are not returned by the ODBC command requesting the list of all tables.

2.4. User-specified Tables

(Used only by generic graphical user interfaces; not needed by users posing direct ODBC SQL queries)

Let C be a category. Let S={ A_1 ,..., A_k } be some unabbreviated attributes of the table C of type Abstract-object (i.e. no attribute A_i ends with an actual concrete attribute of an original semantic category). (Recall that the name of C is a prefix of each A_i).

We define a virtual table T(S) as the projection of the table C on the of attributes SPP comprised of the attributes S, their prefixes, and one-step extensions of the prefixes.

(An attribute A is a prefix of an attribute in S iff A is in S or A_w is in S for some string w. An attribute B is a one-step extension of an attribute A iff B=A or B=A__w where w contains no underscores.)

The name of T is generated as follows: for each A_i let B_i be the shortest synonym of A_i. The name of T is: B1____B2____....Bk

2.5. Semantics of Updates

Release 0.2 supports only restricted updates:

delete from C **where** condition Removes objects from the root category C (does not delete them from supercategories of C).

insert into C attributes **values** assignments

Creates a new object, places it in the root category C, and relates it to some one-step attributes (i.e. the original attributes/relations of category C and their inverses.)

insert into C attributes query

Evaluates the query, resulting in a set of rows. For each row, a new object is created and placed in C. It's one-step relationships are assigned values from the rows.

update C **set** A_1=e_1,...,A_k=ek **where** condition

Selects a set of object of category C. For each of them updates some one-step attributes. For example, to make a person become a student: **update** PERSON **set** STUDENT=PERSON **where** *condition*. To move the person from subcategory of students to subcategory of instructors: **update** PERSON **set** STUDENT=null,INSTRUCTOR=PERSON

insert into C__R ...

Allows creation of multiple relationships R. This cannot be accomplished with previous commands when R is many-to-many.

delete from C__R **where** condition

Allows deletion of multiple relationships R.

3. EXAMPLES OF SEMANTIC SQL AND COMPARISON TO RELATIONAL SQL

This section contains: the semantic schema of a Hydrology application; a normalized relational schema of the same application (a real schema, not our virtual schema); several SQL statements written for the semantic schema and (for comparison) for the relational schema.

3.1. Hydrology Application, Semantic Schema

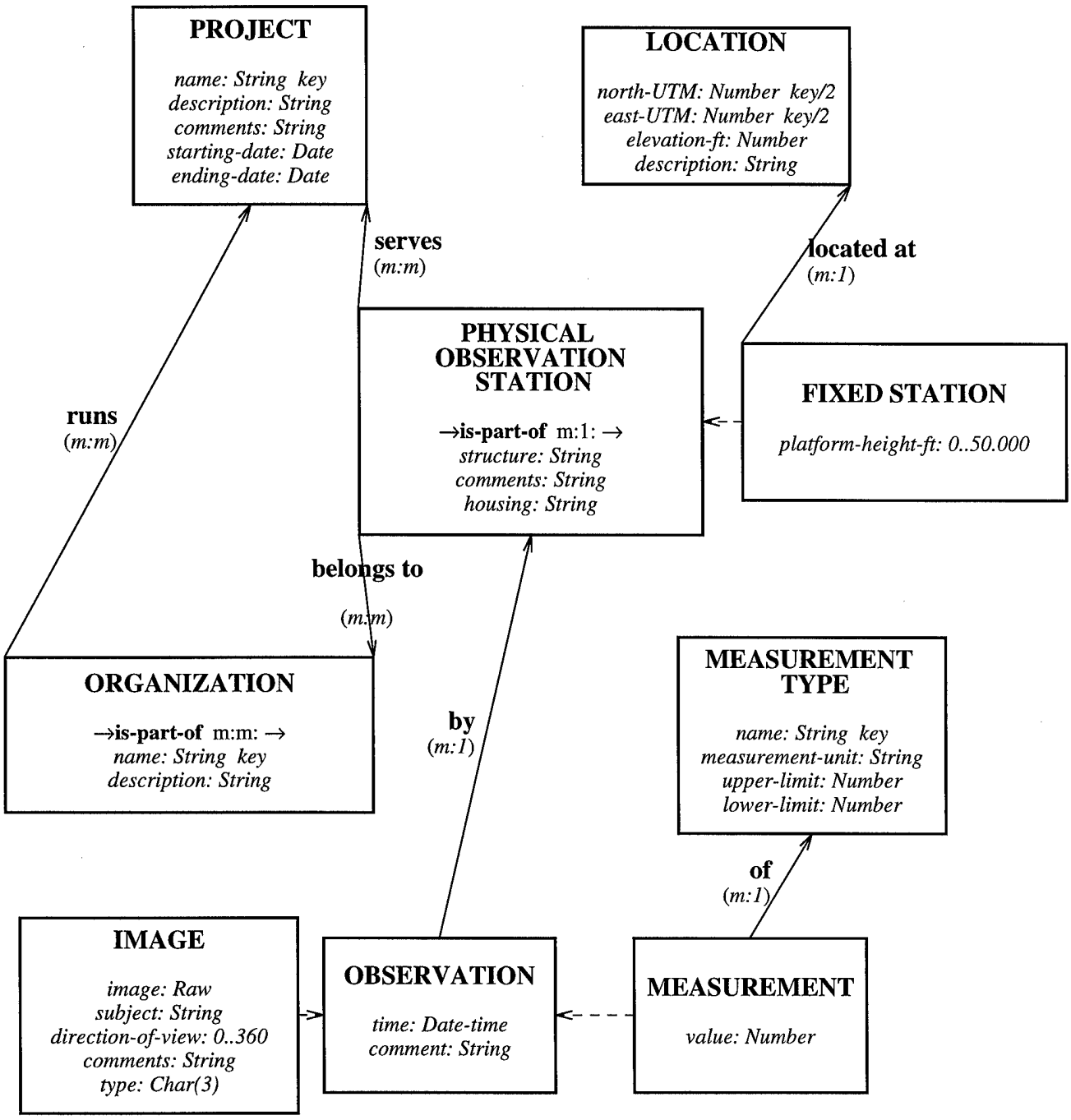


Figure 3-1. Semantic sub-schema for physical observations.

3.2. Relational Schema of the Hydrology Application

PHYSICAL-OBSERVATION-STATION

physical-observation-station-id-key:Integer 1:1; *comments*:String; *housing*:String; *structure*:String; *is-part-of--physical-observation-station-id*:Integer;

LOCATION

north-UTM-in-key:Number; *east-UTM-in-key*:Number; *elevation-ft*:Number; *description*:String;

ORGANIZATION

name-key:String 1:1; *description*:String;

PROJECT

name-key:String 1:1; *description*:String; *comments*:String; *starting-date*:Date; *ending-date*:Date;

MEASUREMENT-TYPE

name-key:String 1:1; *measurement-unit*:String; *upper-limit*:Number; *lower-limit*:Number;

FIXED-STATION

physical-observation-station-id-key:Integer 1:1; *platform-height-ft*:0..50.000; *located-at--north-UTM*:Number; *located-at--east-UTM*:Number;

MEASUREMENT

observation-id-key:Integer 1:1; *comment*:String; *time*:Date-time; *value*:Number; *of--name*:String; *by--physical-observation-station-id*:Integer;

IMAGE

observation-id-key:Integer 1:1; *comment*:String; *time*:Date-time; *image*:Raw; *subject*:String; *direction-of-view*:0..360; *comments*:String; *type*:Char(3); *by--physical-observation-station-id*:Integer;

Figure 3-2. Relational sub-schema for physical observations. Part I: tables representing the categories.

PHYSICAL-OBSERVATION-STATION--BELONGS-TO--ORGANIZATION
physical-observation-station-id-in-key:Integer; *organization--name-in-key*:String;

ORGANIZATION--RUNS--PROJECT
organization--name-in-key:String; *project--name-in-key*:String;

PHYSICAL-OBSERVATION-STATION--SERVES--PROJECT
physical-observation-station-id-in-key:Integer; *project--name-in-key*:String;

ORGANIZATION--IS-PART-OF--ORGANIZATION
organization--name-in-key:String; *organization-2--name-in-key*:String;

Figure 3-3. Relational sub-schema for physical observations. Part II: tables representing the m:m relationships.

3.3. Program Size Comparisons: SQL

1. List of the time and housing of temperature measurements over 50 degrees

SQL statement based on semantic schema:

```
select housing,time from MEASUREMENT where of__name='Temperature' and value>50
```

SQL statement based on relational schema:

```
select housing, time
from PHYSICAL_OBSERVATION_STATION, MEASUREMENT
where exists
    (select * from MEASUREMENT-TYPE
    where name_key = of__name and name_key = 'Temperature' and
        by_physical_observation_station_id = physical_observation_station_id_key and
        value > 50)
```

2. The descriptions of organizations and locations of their fixed stations

SQL statement based on semantic schema, Alternative 1:

```
select description, belongs_to___located_at__LOCATION from ORGANIZATION
```

SQL statement based on semantic schema, Alternative 2:

```
select description, LOCATION from ORGANIZATION
```

SQL statement based on relational schema:

```
select description, LOCATION.north_UTM_in_key, LOCATION.east_UTM_in_key
from ORGANIZATION, LOCATION
where exists
    (select * from FIXED_STATION
    where exists
        (select *
        from
            PHYSICAL_OBSERVATION_STATION__BELONGS_TO__ORGANIZATION
        where name_key = organization__name_in_key and
            PHYSICAL_OBSERVATION_STATION__BELONGS_TO__ORGANIZATION.
                physical_observation_station_id_in_key =
                FIXED_STATION.physical_observation_station_id_key and
            located_at__north_UTM = north_UTM_in_key and located_at__east_UTM =
                east_UTM_in_key ))
```

3. The observations since January 1, 1993 (including images, measurements and their types) with location of the stations

*SQL statement based on **semantic** schema:*

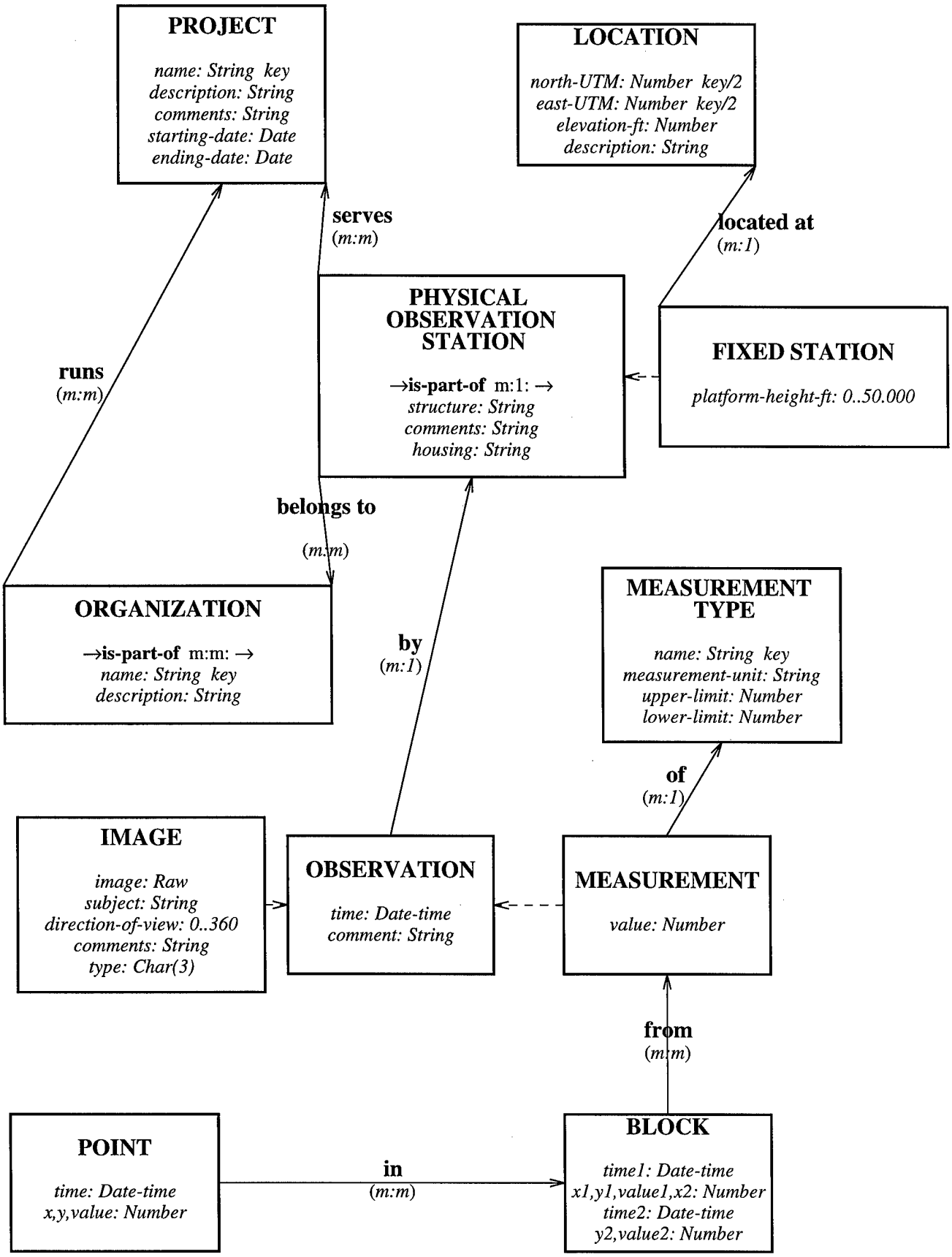
```
select OBSERVATION__, of__, LOCATION from OBSERVATION where time>'1993/01'
```

*SQL statement based on **relational** schema:*

```
(select MEASUREMENT_TYPE.*, LOCATION.north_UTM_in_key,
    LOCATION.east_UTM_in_key, MEASUREMENT.*, NULL, NULL, NULL, NULL,
    NULL, NULL, NULL, NULL, NULL

from MEASUREMENT_TYPE, LOCATION, MEASUREMENT

where time > '1993/01' and exists (select * from FIXED_STATION where
    by__physical_observation_station_id = physical_observation_station_id_key and
    located_at__north_UTM = north_UTM_in_key and located_at__east_UTM =
    east_UTM_in_key and of__name = name_key )) union

(select MEASUREMENT_TYPE.*, NULL, NULL, MEASUREMENT.*, NULL,NULL,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL

from MEASUREMENT_TYPE, MEASUREMENT

where time > '1993/01' and not exists (select * from FIXED_STATION where
    by__physical_observation_station_id = physical_observation_station_id_key and
    of__name = name_key )) union

(select NULL, NULL, NULL, NULL, LOCATION.north_UTM_in_key,
    LOCATION.east_UTM_in_key, NULL, NULL, NULL, NULL, NULL, NULL,
    IMAGE.*

from LOCATION, IMAGE

where time > '1993/01' and exists (select * from FIXED_STATION where
    by__physical_observation_station_id = physical_observation_station_id_key and
    located_at__north_UTM = north_UTM_in_key and located_at__east_UTM =
    east_UTM_in_key )) union

(select NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
    NULL, IMAGE.*

from IMAGE

where time > '1993/01' and not exists (select * from FIXED-STATION where
    by__physical_observation_station_id = physical_observation_station_id_key))
```

3.4. Handling of Interpolated Spatial Functions

The following userview contains infinite virtual categories POINT and BLOCK



Query1: temperature value(s) of a given point

SELECT value FROM POINT WHERE name="Temperature" and <time,x,y>=...

Query2: areas that had temperatures between 1 and 1.1 degrees

SELECT time1,time2,x_1,x_2,y_1,y_2

WHERE name="Temperature" and value>= 1 and value<=1.1

4. APPENDIX: SURROGATES

Object surrogates

For some categories in the schema, our main userview contains surrogates, which are strings that identify objects of that category. These surrogates are computed virtual attributes; they are used in our database languages, e.g. the Semantic SQL.

The database schema defines semantic keys for some categories. A semantic key of a category C is a set of relations and attributes of C that when they all exist (non-null) jointly identify the objects of a category. To compute a surrogate from a semantic key we concatenate the values of the key attributes and relations of an object, replacing any abstract objects in the key relations by their surrogates if the latter exists (otherwise the whole surrogate is null).

For each category C having surrogates, the latter are represented in a virtual attribute:

- C-*id* — attribute of C, range: *String* (*1:1*)

It is formally defined as follows.

Let k(C) be the semantic key of C if one is defined in the semantic schema. (If k(C)=(R_1, R_2, ... Rn) it means that there is an integrity constraint:

for every c_1, c_2 in C:
for every x_1,..., xn in OBJECT:
if c_1 R_1 x_1 and ... and c_1 Rn xn and c_2 R_1 x_1 and ... and cn Rn xn
then $c_1 = c_2$

(The relations R_i do not have to be total, unlike keys of relational databases; nor do they have to be attributes; they have to be m:1 or 1:1.)

We define auxiliary concatenation operator x|y: If x or y is null then the result is null. Otherwise, the result is concatenation of x and y separated by the character '/'. E.g. 'abc'|'cde'='abc/cde'.

The surrogate of an object x, s(x), is defined as follows (null if any part is undefined):

If x is a string then s(x)=s.

If x is a concrete object other than string (number, Boolean, date) then s(x) is a conversion of x into a string.

If x is and abstract object which belongs to only one category, C, for which a semantic key is defined in the schema, then:

Let (R_1, ... , Rn) be the alphabetical ordering of the semantic key of C.

s(x)= s(x.R1)|s(x.R2) ... s(x.Rn)